Layered Architecture for Automatic Generation of Conflictive Animations in Programming Education

Andrés Moreno, Mike Joy, Niko Myller, and Erkki Sutinen

Abstract—Fundamental concepts of programming and data structures are usually taught with graphical tools such as simulations and animations. Conflictive animations have been proposed to improve students' understanding of programming concepts. In conflictive animations, errors are introduced in the animations to motivate students to constantly check their knowledge against what is being animated. We have implemented a framework in an animation tool that allows the automatic generation of conflictive animations of statements, expressions, and other programming constructs. The automatic generation is challenging due to the alternative paths execution can take and their side effects. The architecture of the tool consists of several layers that can alter the normal interpretation or visualization of the program. The framework and the tool have been evaluated by creating conflictive animations of two programming concepts—for-loops and inheritance—and by running a set of 27 examples taken from Java textbooks. Of these, over two thirds (19) required no modification or only minor changes to create the conflictive animations. The reasons that the remaining examples did not generate conflictive animations automatically were divided between the layered architecture used and the example program itself.

Index Terms—CS1, animation, programming education, conflictive animation, object-oriented programming, Java, errors.

1 Introduction

Conflictive animations form a new approach to the use of animations in programming education [1]. Conflictive animations are created so that they do not animate faithfully what the programs intend to do. They aim to compel the student to review the animation critically by asking them to identify possible errors or mistakes in the animation.

Previously, program animation has been used to demonstrate the fundamental concepts of programming. Teachers present programming concepts along with their visual representation using program animation tools. Programs written by teachers or students are animated step by step showing how the computer executes its statements. Teachers can concentrate on the explanations as the tool provides the correct graphical representation. Students can later use these same tools to review the lessons or debug their own programs.

It is expected that students construct their knowledge as they write and visualize their own programs. Ben-Bassat Levy et al. [2] found that animations helped the "mediocre" students to express their knowledge and communicate with the teacher, but that weak and strong students did not benefit as much. Their study was carried out in a high school, and they emphasized that animations should be explicitly taught and explained, as they are not usually self-explanatory.

In another study, Moreno and Joy [3] used an animation tool (Jeliot 3) to support the lectures and assignments of a project-based course at a university. In this setting, they found that students often failed to use the tools properly and to understand the concepts represented in the animation. When asked about complex concepts such as object construction, students could not explain the steps depicted by the animation, but they still regarded the animation as useful. Their pragmatic approach to using the program animation tool was nonetheless useful for creating simple programs, since they used it as a debugger. However, the students appeared not to be making an effort to understand the concepts being animated, such as variables or loops.

When students visualize conflictive animations and engage in detecting the errors, the intention is that they carefully follow the animation and check their knowledge and lecture notes with what happens on the screen. This kind of activity should motivate students to better understand programming concepts and their detailed implementation.

Common misconceptions that students have can potentially be corrected if students pay more attention to animations. For example, Sorva described misconceptions students held about variables in programming [4]. Program animations (in Jeliot 3) provide a self-contained visual explanation of what variables are and how they behave; however, the meaning may not be apparent unless the students observe the animation carefully.

Hundhausen et al. [5] reviewed the evaluations of algorithm animation tools found in the literature. Their metareview concluded that animations were usually positive for students' attitudes, but not always beneficial for

• A. Moreno, N. Myller and, E. Sutinen are with the Department of Computer Science and Statistics, University of Joensuu, PO Box 111, 80101 Joensuu, Finland.
E-mail: {amoreno, niko.myller, erkki.sutinen}@cs.joensuu.fi.
• M. Joy is with the Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK. E-mail: M.S.Joy@warwick.ac.uk.

Manuscript received 29 Apr. 2009; revised 29 June 2009; accepted 31 July 2009; published online 20 Aug. 2009.
For information on obtaining reprints of this article, please send e-mail to: lt@computer.org, and reference IEEECS Log Number TLT-2009-04-0086.
Digital Object Identifier no. 10.1109/TLT.2009.36.

learning. They suggested that the importance of the animation is not what is animated, but how students interact with it, revealing the importance of engagement in animation tools. Thus, the lack of engagement is one of the problems found in current animation tools. Naps et al. [6] acknowledged this and developed a taxonomy of engagement for visualization tools consisting of six main levels of engagement: viewing, responding, changing, constructing, and representing.

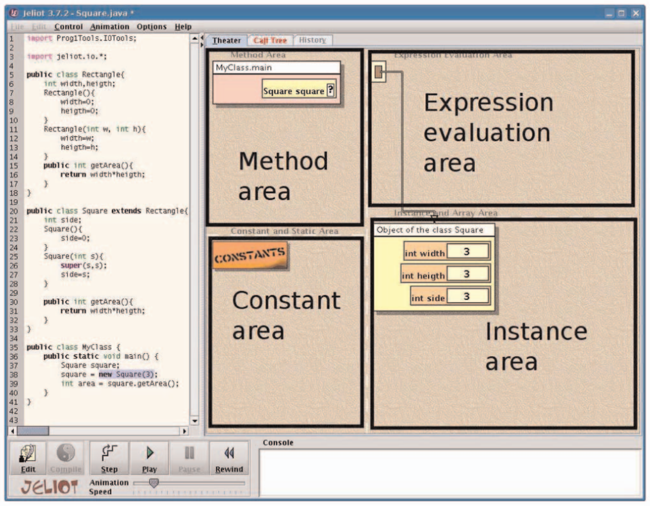
When using animation tools, teachers often complain about the time-consuming task of creating animations of their own examples [7]. Ihantola et al. reviewed the degree of *effortlessness* of visualization tools [8]. Their taxonomy includes three categories—scope, integrability, and interaction—which were deemed important by the surveyed educators when adapting visualization tools to their learning practice. The *scope* category evaluates whether the tool can be used in a single lecture, for a whole course, within a computing domain, or whether it does not impose any limit to where it is used. *Integrability* refers to the ability of the tool to integrate with the teaching environment, materials, and practices, e.g., the ability to run in several platforms, or for the animations to be customized. Finally, *interaction* is divided into two subcategories: the producer-tool interaction (time required to prepare a task with the tool), and the user-tool interaction (ways a user can interact with the produced task following the taxonomy described by Naps et al. [6]).

This paper presents a framework for creating conflictive animations, and describes its implementation in an existing programming animation tool, Jeliot 3. The resulting new tool has been named *Jeliot ConAn*, for conflictive animations. Adding conflictive animations to Jeliot 3 should increase the level of engagement of the student in an effortless manner for the instructor, so that conflictive animations can be used within the course scope, and with no changes to the instructor's examples. Varied programming concepts, such as statements or expressions, are a possible source for conflicts in the framework. Jeliot 3 has a layered and interpretative architecture which provides the means to generate the conflictive animations for those concepts automatically.

First, the topic of programming animation and errors in education is presented, and Jeliot 3 is described in Sections 3 and 4. The following sections categorize conflictive animations and propose a framework to implement them in current program animation systems. An actual implementation of the framework in Jeliot 3 is described in Section 7, and both the framework and its implementation are evaluated in Section 8 according to its technical and effortlessness merits.

2 Related Work

The concept of automatic generation of conflictive animations merges two previously separate threads—*program animation* tools and *errors in education*—which have not usually been considered together before. Program animation tools can automatically generate explanatory animations of programs, and these animations are usually correct as they have been used as means to convey knowledge. On the other hand, errors in education have been proposed as a way to engage students to become critical and deep thinkers [9].



```
import Prog1Tools.IOTools;

import jeliot.io.*;

public class Rectangle{
    int width,heigth;
    Rectangle(){
        width=0;
        heigth=0;
    }
    Rectangle(int w, int h){
        width=w;
        heigth=h;
    }
    public int getArea(){
        return width*heigth;
    }
}

public class Square extends Rectangle{
    int side;
    Square(){
        side=0;
    }
    Square(int s){
        super(s,s);
        side=s;
    }

    public int getArea(){
        return width*heigth;
    }
}

public class MyClass {
    public static void main() {
        Square square;
        square = new Square(3);
        int area = square.getArea();
    }
}
```

Fig. 1. User interface of Jeliot 3.

In this section, we briefly describe the state of the art in programming animation tools, with special focus on how they are implemented. We also introduce how errors have been used up to now in education, especially in programming education.

2.1 Program Visualization

Automatic generation of program animations is possible with tools like Jeliot 3 [10], Problets [11], and WinHipe [12]. Algorithm animation tools like MatrixPro [13] allow the automatic creation of animations of certain data structures and operations on them.

WinHipe animates functional programs written in the Hope language. The animation consists of a step-by-step execution of the program showing how functional statements and expressions are evaluated. The animation displays the trace of the function calls in a tree. Each node in the tree represents a function call and contains the value of the parameters it has been called with. The animation is produced by a functional interpreter that generates each image after every step in the interpretation, resulting in an efficient but highly coupled implementation. Different visualization paradigms, or modifications to the current one, would require modifications to the Hope interpreter.

Jeliot 3 allows the user to visualize Java programs by producing a step-by-step animation of the program flow as it is executed by a Java interpreter. In this case, the interpreter and the animation are separated. The interpreter produces an intermediate code that is used to later direct the animation, and this is discussed in more detail in Section 3 and Fig. 1.

Kumar and Kasabov developed Problets [11], which are randomly generated C++ programs that include an animation and an explanation of its execution. Rather than focusing on the program flow as Jeliot 3 does, Problets' animations focus on the *data space*, animating the changes to values of variables and changes to the execution stack. The implementation of Problets completely separates the animation from the execution using an *Observer architecture*. In the Observer architecture, the visualization engine observes when the values of structures relative to the executed

TABLE 1
Summary of Features of Automatic Program Animation Tools and MatrixPro

Tool	Paradigm	Architecture	Data Flow	Control Flow	Multiple visualization	Backward execution
WinHipe	Functional	Interpreter	Yes	Yes	No	Yes
Jeliot 3	OOP	Interpreter	Yes	Yes	Yes	No
Problets	OOP	Observer	Yes	No	Yes	Yes
MatrixPro	Algorithms	Model-View-Controller	Yes	No	Yes	Yes

program (variables, method stack, . . .) change, and then it updates the graphical representation accordingly.

Finally, the MatrixPro environment focuses on animating data structures and algorithms [13], such as sorting an array. Users can enter their own data sets and watch how the algorithm performs its operations in the selected data structure. The implementation of MatrixPro is based on the Model-View-Controller architecture, similar to the Observer architecture. This gives the necessary flexibility to visualize the same algorithm in different data structures.

Table 1 summarizes the main features of these environments. The last two columns specify whether the tool and its architecture provide support for generating *multiple animations* of the same source code, and whether an animation can go *backwards* seamlessly. In our case, we focus on creating tools for imperative or object-oriented programming (OOP), thus WinHipe and MatrixPro are not relevant. Both Problets and Jeliot 3 can provide automatic generation of program animations for the data space or data flow; however, the ability of Jeliot 3 to animate the control flow of a program makes it better suited to produce conflictive animations relative to those concepts (*if* statements, loops, . . .).

2.2 Errors in Education

Traditional education has focused on conveying correct information to students. A constructivist approach, on the other hand, views students as the ones *constructing* their own knowledge with the guidance of tutors [14]. In this setting, expressing oneself and making errors is encouraged. The process of debating about the concepts, and the different perceptions of them, helps students to form their own understanding. Postman argues that students would be more engaged in their own education if they were shown that people fail, make errors, and if they were encouraged to identify those errors [9].

Große and Renkl studied how fixing errors in incorrect worked examples could foster learning in physics education [15]. According to the study, far transfer of knowledge improved on those students with prior knowledge that used both correct and incorrect worked examples. The possible explanation given for this improvement is that "the contrasting features (here, the error) attract attention and elicit related learning processes." Moreover, students correcting the errors were not given any feedback on how to correct the error, and Große and Renkl suggest the use of a computer-based learning environment to link the incorrect and the correct solutions to provide the student with a self-assessment of their explanation.

Exercises that use incorrect code are common in programming education. Students are often asked to find and correct syntax or design errors [16]. These kinds of exercises are usually simple and do not assess the understanding of the dynamic behavior of programs. At different level, the MatrixPro algorithm simulation tool [13] contains an activity in which students are asked to work with a faulty implementation of a binary search tree. Students have to graphically add keys to the tree that will result in breaking one of the properties of such trees. This particular activity assesses not only whether the student understands how the correct implementation works, but also evaluates what makes a faulty binary tree. At the time of writing there is no published report on the effectiveness of this kind of activity.

In their taxonomy of "visual algorithm simulation exercises" [17], Korhonen and Malmi describe this MatrixPro activity as a "complete open question," where all of the exercise components—the algorithm used, its input, and its output—are explicitly questioned. In the exercise, the algorithm would be devised by the student to be faulty. The student freely chooses input to simulate an algorithm that will result in a faulty output, a broken binary search tree. With this kind of activity, Korhonen and Malmi implicitly add a correctness dimension to their taxonomy, but it is not developed further.

Cognitive conflict activities, a technique commonly used in physics education, put the students in situations where they are faced with their misconceptions. According to cognitive conflict theory [18], students are then ready to accept a correct understanding as their own previous understanding has been shown to be not valid. In a recent study, Ma et al. [18] used cognitive conflict to solve comprehension issues with variable assignment. Students who held misconceptions were shown animations of correct assignments. Ma et al. found that cognitive conflict and animation help students to resolve misconceptions.

Ma et al. used animations to resolve a previously identified misconception or conflict. However, in our case, the animations will also create the conflicts automatically and students will be the ones self-assessing their knowledge and understanding.

3 Jeliot 3

In order to demonstrate the capabilities of conflictive animations in learning programming, we have used the Jeliot 3 program animation environment, which has been shown to be effective in improving the learning of elementary computer science and programming [2]. It visualizes Java programs automatically without any user involvement, and thus novices can start using Jeliot 3 on their first day of learning to program. Jeliot 3 supports the addition of "stop and think" questions, which ask the students about the result of executing the following statement or expression.

The user interface of Jeliot 3 is illustrated in Fig. 1. The source code editor is in the left-hand pane, while the right-hand pane is used to display the animation. VCR-like buttons to control the animation are located in the lower left corner. Fully dynamic animation of the data and control flow of the program is displayed, including every aspect of the program execution (e.g., method calls, object construction, and expression evaluation). The animation is created automatically from the source code, so that the student needs only to learn to use the control buttons in order to work with the tool.

Jeliot 3 requires each program to be assembled in one file, even if it contains several classes, before it is animated. A further restriction is the use of the libraries that are supported by Jeliot 3.

An animation in Jeliot consists of a step-by-step execution of the program and no step is omitted, and students can thus match every single line of code with its actual execution through the animation.

The Jeliot 3 animation pane is divided in four main areas: method area, evaluation area, constants area, and instance area. Method calls result in frames displayed in the method area, and these frames contain the local variables, which can hold primitive values or references to objects and arrays located in the object area. When an expression is to be evaluated (e.g., "$\mathrm{a}+\mathrm{b}$"), the values that the expression is composed of move from the method frame, object area, or constant area to the evaluation area, where the operation is executed and the result shown. In Jeliot 3, control flow statements (*if* statements, *for* loops, ...) are animated in detail, and the user is informed of what is going to happen next (e.g., the else branch will be executed). Fig. 1 shows in the animation pane a reference to an object located in the instance area that is going to be assigned to a variable in the method frame. This object contains three primitive values.

Jeliot 3 has the potential to create conflictive animations either by altering the data flow, for example, by changing the values of the variables, or by altering the flow of the execution, such as by exiting the *while* loop when the condition is true. In Section 5, we present in more detail the possible levels of conflictive animations which can be applied in programming education.

4 Architecture

One of the architectural goals of Jeliot 3 was to make it modular so that it could easily be integrated with other systems, and new packages or subsystems could be integrated with it [19]. Furthermore, this allowed the use of components of the previous Jeliot versions (for example, the animation engine of Jeliot 2000 was reused with only minimal modifications).

The interpretation and animation of Java programs are separated into two modules in Jeliot. For the former, DynamicJava,[1] an open source Java interpreter, processes the user program. In the latter module, Jeliot's graphical engine creates an animation of the program interpretation. To connect these parts, an intermediate code, MCode, was designed.

1. http://koala.ilog.fr/djava/.

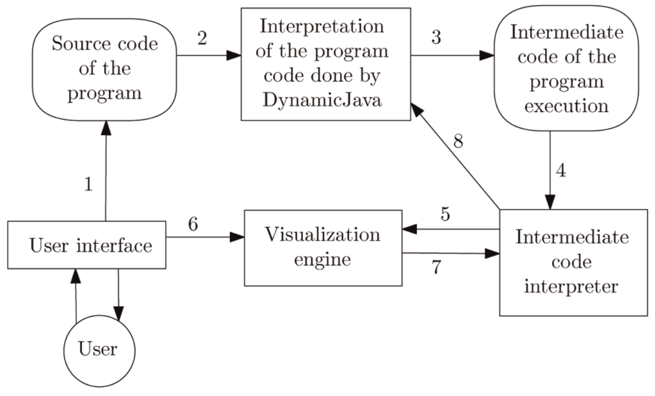

Source code of the program
Interpretation of the program code done by DynamicJava
Intermediate code of the program execution
User interface
Visualization engine
Intermediate code interpreter
User
1
2
3
4
5
6
7
8

Fig. 2. The functional architecture of Jeliot 3.

MCode [20] is a textual representation of the interpretation of a running program, or a program trace. It not only describes the changes in variables and stacks, as a normal Java debugger would do, but it also details the operations that produced those changes. All this information is required to animate every step in the execution of a program. At present, there are two MCode interpreters in Jeliot 3: One that produces the complete animation of the program and another one that builds the method calls tree during the execution. Program animation designers can build their own animations and integrate them into Jeliot 3 by writing a new MCode interpreter. MCode aims to be language independent—for example, a version of Jeliot has integrated a Python interpreter to produce MCode for Python programs, which can be now visualized with Jeliot 3.

The functional structure of Jeliot 3 is shown in Fig. 2. A user interacts with the user interface and edits the source code of the program (1). The source code is sent to the Java interpreter and the intermediate code is extracted from the interpretation (2 and 3). The intermediate code is then interpreted (4) and directions are given to the animation engine (5). The user can control the animation by playing, pausing, rewinding, or playing step by step the animation (6). Furthermore, the user can input data (6), numbers, or strings, to the program executed by the interpreter (7 and 8).

The intermediate language provides a source of interpretation information that can be used for different visualizations. A new intermediate code interpreter and an animation engine can be developed to produce a different visualization of the same program (such as call tree visualization). Thus, Jeliot 3 can be extended internally with multiple visualizations of the same program.

Readers interested to get more specific information about the design of Jeliot 3 and the MCode intermediate code are referred to Myller [19] and Moreno [20] for detailed explanations of Jeliot 3 development.

The architecture of Jeliot 3 allows the production of conflictive animations at several of its layers, from source code to interpretation to animations. Thus, it is a good platform to highlight the possibilities of conflictive animations. These different layers will be analyzed in Section 6

TABLE 2
Summary of Programming Concepts and Examples of Conflictive Animations

Concept	What	Example of conflict	Possible in Jeliot
Module	Program level concepts like algorithms	Animating a different program	No
Statement	Assignments and flow control structures	Altering behaviour of loops	Yes
Expression	Binary and unary operations. Method calls and object creation	Calling an incorrect method	Yes
Operator	Logical and arithmetic operators	The opposite operator is visualized	Yes
Literal	Constant values in variables, expressions	Converting the numerical value to a less precise type, from double to int	Yes

discussing their possibilities and drawbacks of producing conflictive animations.

5 Programming Concepts in Conflictive Animations

Moreno et al. [21] present five different levels of conflictive animations for programming. These levels were derived from the main production rules of the Python abstract grammar,[2] which is simpler than the Java abstract grammar. However, these five main levels (module, statement, expression, operators, and literals) correspond to the fundamental building blocks of Java and other programming languages.

Object-oriented concepts are indirectly addressed at the module and expression levels. Classes relate to modules and message passing and object constructions relate to expressions. It should be noted that this categorization does not consider software design issues that are central to object-oriented programming.

The possibility of automatically generating conflictive animations in Jeliot 3 for the concepts presented by Moreno et al. [21] are discussed in the following paragraphs. Table 2 summarizes the possibilities of Jeliot 3 to produce conflictive animations of the concepts described.

5.1 Module

Conflictive animations at this level animate either a completely different program or a different implementation of a class from the one the user provides—that is, the module is replaced before being animated. Automatic generation of such a conflictive animation is not possible in Jeliot 3, since it only uses the source code of the program and cannot make inferences on what will be a good program with which to replace the existing one. Jeliot 3 could give the educator the option to select two programs, one that would be animated and one that would be displayed in the code editor pane. However, this would lead to problems as the code highlighting cannot highlight the displayed source code. This level is better suited for algorithm animation, where a data structure could be substituted by a different one or by a faulty one, such as the faulty binary tree used in MatrixPro, as discussed in Section 2.1.

2. http://docs.python.org/lib/Python.txt.

5.2 Statement

Statements include assignments and control structures like loops and *if* statements. Automatic generation of conflictive animation of statements is possible in the current design of Jeliot 3. Interpretation of statements can be changed to produce incorrect animations of their execution. This new interpretation requires careful planning so that it is close to the original one and reflects possible misconceptions students have with those statements. For example, an incorrect animation of a *for* loop could execute the initialization of the counter in every iteration. The incorrect animation should not be *too* similar to the original, because there is the risk that the resulting conflict cannot be perceived in the animation by the student.

5.3 Expression

Logical and mathematical expressions are not exclusive to programming, and we tend to assume that students know about them. However, if Jeliot 3 is used as a debugger, simple expressions like string concatenation are usually overlooked by students [3]. Simple conflictive animations can be automatically generated in Jeliot 3 to test students' attention and to make them realize that an expression can also be a source of problems in their own programs. The conflictive animation will just interpret operators differently. For example,"<" behaves like a ">" or "<= ." These conflictive expressions can produce a conflictive behavior of a loop or an *if* statement if they are used within a condition. Such a conflictive behavior of a loop or *if* statement can make the student think that the source of the error is somewhere else than in the execution of the conflictive expression.

Jeliot 3 also shows the steps taken with complicated expressions involving increments and decrements, both in the pre (`++a`) or post (`a++`) form. Identifying incorrect animations of increments is an activity that could lead to a better understanding of such concepts. Two other main concepts are also considered as expressions.

Function calls. These are key building blocks for programming and together with statements they determine the flow of a program. Automatic conflictive animations can be generated by altering the name resolution that is inherent to Java. This way a method call can be conflictively animated by actually executing a method with a signature different from the original method call.

Instance creation. Object-oriented programs rely on creating objects. However, object creation is a difficult concept as it involves several steps that are not explicit in the code [22]. Jeliot 3 animates those steps and can create an automatic conflictive animation of them. It would involve either changing the order those steps are animated or eliminating some or all of them.

5.4 Operator

The operator level is tightly linked with the expression level. At this level, however, operators will not change their meaning or interpretation, but they will change their graphical representation in the animation. For example,"$3 < 4$" may show like "$3 > 4$" or "$3 <= 4$," and the result will be "true," the correct one for "$3 < 4$." In this case, the program flow will continue as it was intended, but the animation was wrong for the brief time the operator appeared on the screen. As the result of the expression not being consistent with what the shown operator indicated, we call this type of animation *nonconsistent*. Generating this kind of conflict is trivial, as it only requires changing the animation of the operator. This level should be used to introduce students to conflictive animations as it requires attention to detail rather than correct understanding.

5.5 Literal

The literal level is the lowest and, like the operator level, conflictive animations at this level will not demand much knowledge from the student. Automatic conflictive animations in Jeliot 3 at this level may focus on the data types. Conflictive animations of values will change the representation of numbers, so that, for example, an integer can be represented as a real, or vice versa.

5.6 Summary

Of the five programming concepts or levels presented here, for four of them it is possible to conflictively animate them in Jeliot 3. However, *statement* and *expression* are the concepts that have an influence on the execution or interpretation of the program, and it is at these levels where we can find the concepts needed for object orientation and program flow. Thus, we consider that being able to automatically create conflictive animations at these two levels will be beneficial for students.

Technically, these two levels are the most challenging to implement as their conflictive interpretation can produce side effects that are neither supported nor expected by traditional animation tools, since such tools are only prepared for the correct execution of a program.

6 Layered Generation of Conflictive Animations

The previous section introduced the possibilities for conflictive animations of different programming concepts. When it comes to the actual generation of conflictive animations we need to take into account the implementation details of the animation tool. In our case, we have chosen Jeliot 3, since its architecture allows for several ways to implement the examples given in the previous section. This section analyzes the layers that create animations in Jeliot 3 and explores the possibilities for them to be part of the conflict creation process.

As said before, animations in Jeliot 3 are generated from the Java source code to be animated. This code is interpreted by DynamicJava, which parses the source code and creates the syntax tree, and then interprets the code by traversing the tree. This interpretation creates the intermediate code, which is in turn interpreted to create the animation through the engine. We can identify then five main layers or stages that run sequentially and can be the source for conflictive animations (from bottom to top): program source code, Java parser, DynamicJava's tree interpretation, MCode interpretation, and visualization engine.

Automatically generating conflictive animations imply modifying the normal behavior of one or more of the stages of the process. The linear generation of the animation implies that changes in one of the lower layers of the process will propagate up to the animation stage. This propagation requires adapting the levels to the new possible incorrect states. In the original design of Jeliot 3, upper layers relied on the correctness of the information handed by lower layers. Thus, upper levels should know when the interpretation is in a conflictive state to allow for possible deviations from the norm.

Because the process of animating is mostly done in one direction, lower layers will not be aware of conflicts introduced in the upper layers. Thus, the rest of the animation will not be consistent with the conflicts introduced by the upper layer. For example, if the animation layer changes the graphical representation of "$5 < 4$" to "$5 > 4$" it will create a conflictive animation of an operator, but the result of the comparison is still evaluated in the tree interpretation layer ("false"), which is not aware of that change, and the animation will continue as if no conflict has happened.

We reflect on the problems and possible benefits on using these layers. At the end of the section, we summarize the benefits and disadvantages and propose a solution to implement conflictive animations within Jeliot 3.

6.1 Source Code Layer

The first possibility for creating conflictive animations in Jeliot 3 is by transparently modifying the source code before it is interpreted. This altered program could be the source for the conflictive animation, and by doing this, there is no need for other major changes in Jeliot 3 to make conflictive animations happen. However, that will require the addition of a structure which will parse the Java program and that will modify the program without adding syntax and grammar errors.

Adding that structure would result in duplicating the work done by the Java interpreter, which already understands the Java source code, and provides a programmatic way to alter the interpretation tree.

This layer could be used to implement conflictive animations at the module level of programming concepts. Jeliot 3 could let educators specify two source files, one that would be animated in the animation window and that would be conflictive, and another one that would just be displayed in the code pane. This would not be automatic and it would not allow for the automatic spotting of the error, but it could be useful in certain scenarios, such as when creating conflictive animations of operations on data structures where one method is replaced by a faulty one.

6.2 Parser Layer

A conflictive parser could alter the syntax rules of Java. For example, a redefined grammar could change the priority of binary expressions, and all the programs interpreted according to that grammar will evaluate mathematical expressions in the wrong order prompting several conflictive animations.

Two main problems appear if conflicts are generated at this layer. First, the generation of several types of conflicts would require the generation of a specific grammar for each type and the ability to change them at runtime. Also, it would be difficult to know whether the animation is in a conflictive state or not.

Grammars could be designed to be more flexible accepting common mistakes and misspellings. Implementing this feature would allow the possibility of animating students' programs that contain syntax errors. In this case, students could learn not to rely in the compiler when looking for syntax errors, but to carefully write their programs.

6.3 Tree Interpretation Layer

This layer is the one that stores the most of the state information of a running program. The tree structure also stores completely the semantics of Java execution, which allows for "tweaking," or fine tuning, the interpretation of Java statements and expressions.

Tree interpretation actually consists of two interpretations, or visits. In the first one, the static class is resolved, and in the second visit the evaluation is performed. The first visit is used to collect data that could be relevant for conflictive animations of method calls.

It is in the second visit where the main work is done to produce the conflictive animations. Evaluation of expressions or statements, such as "i++," can take two paths: the normal path, where it is evaluated according to the Java rules, or the alternative path, where an incorrect interpretation of the statement or expression is carried out. This incorrect interpretation will produce the MCode that reflects its execution, and this MCode is sent normally to the intermediate code interpreter to generate the animation.

Conflictive interpretation can lead the program to an unexpected halt. This problem could arise when an *if* statement condition is evaluated to "true" when it should be "false." If this condition was guarding a division by zero error, the conflictive animation will display the *Divided by Zero* exception, even when the program was written to avoid that. In such cases, educators should prepare the conflictive animation examples and assignments carefully so the execution of the example will not encounter uninitialized values errors or exceptions.

For complex expressions, such as method calls, the possibility of simultaneous conflicts is disabled. This is done to reduce the number of possible conflicting interactions in the interpretation.

6.4 MCode Interpretation Layer

According to the architecture of Jeliot 3, it is possible to add a new intermediate code interpreter to produce a new visualization of the program, such as a conflictive one. Thus, modifying the program interpretation at this layer would have the benefit that the modifications at this layer will not interfere with the normal behavior of the Java interpreter.

However, the layer lacks the information to be able to decide when and how to produce conflictive statements as most of the context is only dealt with at the previous layer. Moreover, in order to generate an animation that is consistent, the new interpreter should be able to interpret the conflictive statement. As in the source code layer, the modifications needed will replicate much of the work of the Java interpreter.

6.5 Visualization Layer

This layer basically follows the instructions that the MCode interpretation layer generates and it does not have any knowledge about the semantics of programs.

At this level, it is straightforward to create nonconsistent conflictive animations, like the ones described above in the Operator and Literal level. The graphics are changed so that incorrect signs or values are drawn.

At this level, there is insufficient information to decide when it is better to change the representation, so the layer will only animate incorrectly animation occurrences of operators and literals according to predefined rules.

6.6 Summary

The five layers presented here have shown capabilities to be the mechanical creators of conflictive animations. However, in some cases, major additions to the code are required in order to provide the intelligence needed for the creation.

In this analysis, we have found out that the Tree Interpreter layer is the most suitable layer for the task of creating conflictive animation. This layer keeps most of the information about the program that can be used to create new conflicts. Furthermore, changes done to the implementation at this layer will automatically propagate through the upper layers and create the conflictive animation with reduced side effects.

7 Jeliot ConAn

The previous considerations have led us to develop Jeliot ConAn. Jeliot ConAn maintains all the features of Jeliot 3 and adds the possibility of creating animations that contain known errors. The implementation of Jeliot ConAn incorporates a framework for adding support to new programming concepts being conflictively animated. Developers have to alter the behavior of the interpreter and create new *conflict objects*.

Fig. 3 gives an overview of the reworked Jeliot 3 architecture including the conflict generation. Conflict objects are central to Jeliot ConAn as they hold the logic of the conflict and the information needed to present them. Presentation and interaction of the conflict objects with the user are taken care of by the framework. Thus, the framework can be divided into two parts: generating conflicts objects (numbers 1-5 in Fig. 3) and presenting them to the user (numbers 6-10 in the same figure).

7.1 Generating Conflictive Animations

The generation of conflicts starts when the user enters the source code for a program (1) that is sent for interpretation by the conflicting version of DynamicJava. The interpreter produces the intermediate code for the execution. At some point, the interpreter will misinterpret a statement in the program according to preprogrammed behavior—for example, an overridden method may be called instead of the overriding method. This misinterpretation will produce an alternative execution that is reflected in the MCode. The resulting intermediate code is surrounded by specific MCode instructions marking the beginning and the end of the conflictive part (3). At this time, the conflict object will have been created (4) containing all the relevant information

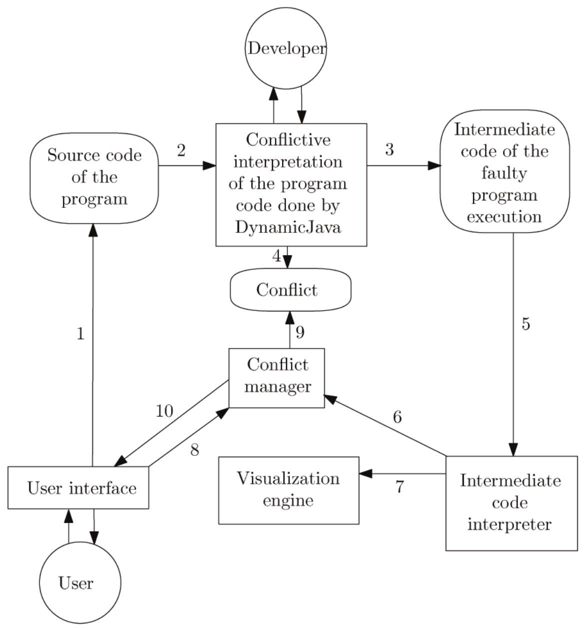

Developer
Source code of the program
Conflictive interpretation of the program code done by DynamicJava
Intermediate code of the faulty program execution
Conflict
Conflict manager
User interface
Visualization engine
Intermediate code interpreter
User
1
2
3
4
5
6
7
8
9
10

Fig. 3. The functional architecture of Jeliot ConAn.

of that conflict (location, method called, class information, etc.). This MCode is sent to the intermediate code interpreter (5), which will interpret the intermediate code line by line as the animation progresses step by step.

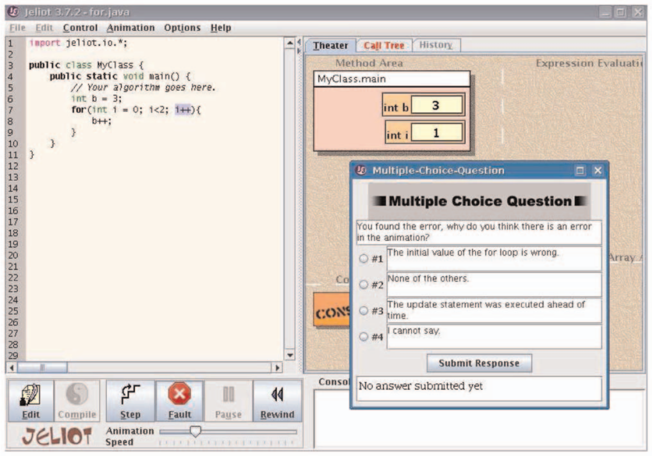
7.2 Presenting Conflictive Animations

When the MCode interpreter receives the instruction marking the beginning of a conflictive part (5), it will report to the conflict manager that the conflict is about to start, or to end (6). The visualization engine will receive the commands to animate normally from the interpreter (7). The last stage of the presentation occurs when the user presses the relevant button and checks for the conflict (8). If a conflict is happening or has happened, the information about it is retrieved (9), and returned to the user through the user interface (10).

7.3 User Interaction

Jeliot 3 user interface has been slightly modified in Jeliot ConAn. As the mission of the student is to identify the error, we include a *Fault Button* which, when pressed, indicates to the animation tool that the user thinks a conflict (i.e., error) has occurred. Moreover, the *Play Button* has been removed to force the student go step by step. See Fig. 4 to see the user interface of Jeliot ConAn when conflictive animations are shown.

Conflictive animations can be used in assessment by asking students to press the *Fault Button* whenever they detect a conflict. To avoid random trials, there is a limit—by default 3—to the number of times the Fault button can be pressed in a single run of the animation. If this limit is reached, the animation is restarted. In addition to this, the conflict may only be apparent to the students a long time after it has happened. In Jeliot ConAn, students have a varying number of steps after which they can still report the occurrence of an error.



```
import jeliot.io.*;

public class MyClass {
    public static void main() {
        // Your algorithm goes here.
        int b = 3;
        for(int i = 0; i<2; i++){
            b++;
        }
    }
}
```

Multiple Choice Question

You found the error, why do you think there is an error in the animation?

#1 The initial value of the for loop is wrong.

#2 None of the others.

#3 The update statement was executed ahead of time.

#4 I cannot say.

Submit Response

No answer submitted yet

Fig. 4. User interface of Jeliot ConAn. In the picture, the student has pressed the Fault button and they are asked why they think there is an error.

When the student presses the Fault button, they are informed of the success of the trial. If unsuccessful, they continue watching the animation looking for an error. If the student has pressed the Fault button at the correct time, a multiple choice question checks whether the reason for pressing it was the correct one. This helps the student to reflect better on what has happened and why, and teachers can collect this feedback to identify misunderstandings held by the students.

Finally, when a conflict animation has been identified, Jeliot ConAn rewinds itself and correctly animates the conflictive concept. If, however, the animation reaches the end and the student has not spotted the error, they are given a hint of where the error is and asked to try again.

7.4 Awareness of the Conflicts

It is still not clear how we should make the students aware of the possible conflicts in the animation. On one hand, students should be aware of the types of conflicts that are possible in the current animation in order to be able to concentrate on the correct fragments of the program animation. Due to the interpretative nature of Jeliot ConAn, it cannot detect and warn in advance of the types of conflicts that are going to happen.

On the other hand, if students are given a clear indication of where to look when the animation is running, the possible benefits from conflictive animations are reduced. In this case, students may not check the knowledge about programming but their knowledge about errors.

8 Evaluation

To evaluate the framework, we have implemented in Jeliot ConAn three conflictive objects. Two of them aim to create conflictive animations related to inheritance topics (*Conflictive Overriding* and *Conflictive Implicit Super Call*) and one is related to the *for* loop (*Conflictive For Update Statement*). These three are just a selection of the possible conflictive animations that could be implemented in Jeliot ConAn.

The selection of these three objects is informed by both technical and pedagogical reasons. Pedagogically, inheritance and object construction are difficult concepts to learn

```
class A{
   public void m(){...}
}
public class B extends A{
   public void m(){...}
   public static void main
            (String[] argv){
      B b = new B();
      b.m();
   }
}
```

Fig. 5. Overriding method call code.

[22], and students have problems describing the processes involved [3]. Technically, the three conflictive objects have been chosen because they show two levels of programming conflicts (expression and statement) which are deemed to be the most important ones, as discussed in Section 5. In our framework, we have relied on the Tree Interpretation layer to automatically generate several kinds of conflictive animations. The selected conflictive objects will indicate if enough information can be gathered at that stage to create conflictive animations.

The evaluation described here aims to test Jeliot ConAn and its framework from a technical point of view—is the framework able to automatically create conflictive animations? If so, how much effort does it require from the developer to implement the conflict, and how much effort from the teacher to use the implemented concepts? To answer both questions, we have first implemented the chosen conflictive objects (see Section 8.1), and then tested whether the objects can create the conflicts from a repository of test programs collected from Java textbooks and online sources, as discussed in Section 8.2. These programs were collected from the chapters where inheritance and *for* loops were explained. The repository contains authentic examples teachers have readily available to explain the concepts.

In this setting, the evaluation can be regarded as positive if 1) the framework allows for the creation of conflictive objects, and 2) the implemented conflict objects can create conflictive animations from independent sources with none or little modification.

8.1 Creation of Conflict Objects

The following implemented conflicts were designed to be automatically created in those situations where the code matched certain conditions. These conditions are different for each conflictive object. Implementing them according to the framework consisted of a series of steps, as follows:

1. Implement a new *Conflict* subclass that can hold the details of the conflict for later reference.
2. Modify the normal tree interpretation of the source code so that it:
   a. creates an object of the Conflict subclass with the information about the particular conflict,
   b. changes the implementation from the original and correct interpretation to an alternative and potentially conflictive one, and
   c. indicates the start and end of the conflictive interpretation using the intermediate code instructions.

```
class A{
   A(){...}
}
public class B extends A{
   int i;
   B(){
      i = 0;
   }
   public static void main
            (String[] argv){
      B b = new B();
   }
}
```

Fig. 6. Implicit super call code.

The second step required more programming effort, in particular, changing the implementation of the interpretation. Object-oriented conflicts proportionally required more work than procedural ones.

We now describe the three conflictive objects implemented in ConAn. We specify what makes them conflictive, and how we have implemented them following the steps described above.

8.1.1 Inheritance: Conflictive Overriding

Conflictive overriding animations are those that animate the execution of the overridden method from a parent class rather than the overriding one which was the one invoked. Thus, the programming concept they are related to is *Expression* level. Fig. 5 shows a code fragment that should activate this conflict. To identify this conflict, the student has to read the source code in advance and to acknowledge that there is an overriding method.

A new class, `ConflictiveOverriding`, was added to the framework to implement this conflictive object. The main fields of the class hold the values for the name of the class and the overridden method. Jeliot ConAn creates the conflict object in the first iteration of the tree interpretation, when the data types and class are resolved, see Section 6.3. It is in the second iteration that the alternative and wrong execution path is taken according to the data contained in the conflict object. The presentation of the conflict allows the student to detect the conflict throughout the time the wrong method is being animated as we consider that not only the call, but also its animation, are part of the conflict.

8.1.2 Inheritance: Conflictive Implicit Super Call

Implicit super calls are added in Java when a constructor does not include a call to its parent class constructor. Fig. 6 shows an example code fragment. This conflict is useful when students are learning about object creation and inheritance in Java. Students have to carefully watch each single step in the animation to identify the missing call to a constructor in the parent class. As the previous one, this conflict belongs to the *Expression* as it relates to method calls.

In the first step, we implemented the class `Conflicti-veImplicit`, which contained variables for the names of a class and its parent class. The second step starts with the construction of the conflict object. This requires correct identification of the existence of an implicit super call, which is done in the first pass of the tree interpreter. It is in the second pass, the actual evaluation, that the implicit super call is evaluated. For Java consistency reasons, we cannot prevent that from happening, but we mark the beginning and the end of interpretation of the super call using the intermediate code instructions. When the intermediate code interpreter receives code relative to the implicit super call, this code is ignored and it is not animated.

8.1.3 Conflictive for Update Statement

When a *for* loop is executed, the update statement is only executed after the loop block has ended. This conflictive object changes the order of execution, and executes the update statement at the beginning. This conflict relates to the *Statement* programming concept. This conflictive animation can be identified straightforwardly if the student knows the behavior of the *for* loop, or maybe later when they acknowledge that the loop has skipped one iteration.

In this case, `ConflictiveForUpdate` is implemented containing only the line where the *for* loop is located. The interpretation of the *for* loop is tweaked in the second pass of the for loop. We changed the order of interpretation of the different parts of a *for* loop. Now, the alternative and potentially conflictive execution will interpret the update statement before the body of the loop. The conflict beginning and end is determined by the interpretation of the update statement.

The presentation of the conflict allows the student to detect the conflict while the update statement is conflictively animated or three steps after its animation.

8.2 Conflictive Animation of Programs

To assess the framework and the implemented conflicts, we have constructed a test set consisting of 27 programs, which are divided into two subsets. One relates to the *for* loop (15 programs), and one to inheritance in general (12 programs). These programs have been gathered from five popular Java books and a website (see the Appendix). The chosen programs are only from those chapters that present the *for* loop, and those which deal exclusively with inheritance. Examples that contained loops or inheritance, but were not trying to explain them, were discarded.

The *for* loop programs (15) were basically variations on two themes: single update statements and complex update statements. The average size of the programs was 20 lines of code. Thus, 15 programs represent a good sample of possible combinations. Inheritance programs (12) presented a wide range of topics related to inheritance, object construction, overriding, abstract classes, and basic inheritance design concepts, and had an average size of 49 lines of code.

The programs have been modified to follow *Jeliot 3*'s conventions, i.e., combining the multiples files in a single file and using the input libraries recognized by Jeliot 3. Moreover, some of the test programs have been extended with the source code required to start executing the program, i.e., adding a *main* method so the program will demonstrate the implemented features.

The results of creating the conflictive animations are divided into four categories, and each program was assigned a category after being tested with Jeliot ConAn. The categories are as follows:

- **Automatic generation**: The program did not require any modification to create the conflictive animation.

TABLE 3
Summary of Testing

Result	Inheritance	For Loop
Automatic generation	3	13
Automatic generation with minor changes	2	0
Automatic generation with major changes	1	0
No generation due to example program	5	0
No generation due to framework	1	2
Total	12	15

- **Automatic generation with minor changes**: The program source code had one or two lines edited or removed to trigger the conflictive animation, e.g., changing a method call in the main method.
- **Automatic generation with major changes**: Important modifications were done to trigger the conflictive animation, e.g., a new constructor was added to the class.
- **No generation due to program**: In this case, it was not conveniently possible to modify the program so that it would trigger the conflict animation. Thus, the animation showed no conflict.
- **No generation due to the framework**: Again, the animation did not show any conflict, or the animation could not be created. However, the framework should have created a conflictive animation because the program contained the elements that should trigger the conflictive animation.

8.2.1 Results

The results of executing the tests in Jeliot ConAn are summarized in Table 3 and described in the following paragraphs. These results are discussed in Section 9.

Of the 12 inheritance programs tested, 6 programs did not produce conflictive animations. The reason is that these programs did not contain any example either of implicit super calls or of overriding methods, and thus the implemented conflicts could not be triggered. They contained basic inheritance examples that demonstrated how classes could extend other classes with new methods or implement *abstract* methods. In one program, Jeliot ConAn failed to animate an overridden method, and this was due to a limitation in the framework as it could not detect the calling object properly.

The remaining six inheritance programs produced conflictive animations for either conflictive overriding or conflictive implicit super method calls. Three of these did not require any change in the source code and two required minor changes. Minor changes consisted of altering one or two lines in the source code. Two of these five programs had the potential to animate both of the possible inheritance conflicts, but the framework only allows for one conflict to be animated per program. One program animated a conflict after a constructor was added to it, and a new object creation had to be added. We considered these to be major changes to the program.

TABLE 4
Engagement Taxonomy and Conflictive Dimension [1]

Level	Well-behaved animation	Conflictive animation	Supported by Jeliot ConAn
Viewing	Students can visualize animations, either step by step or continuously.	Students should be aware of the possibility of viewing an incorrect animation.	Yes
Responding	Students are asked to respond to questions related to an animation and the concepts presented in it.	Students have to detect the errors in the animation.	Yes
Changing	Students change the animation to explain a different concept that the animation was meant for.	Students change and correct a conflictive animation.	No
Constructing	Students use the animation tools to create an animation that explains an algorithm or a simpler concept.	Students purposely create a conflictive animation.	No
Presenting	Students verbally present an animation to an audience.	Students present the conflictive animation and try to confuse peers.	No

Of the 15 *for* loop examples, 13 produced conflictive animations without any modification. The two programs that did not produce conflictive animations used external libraries that Jeliot could not access, such as *java.util.**, and thus animation could not be generated. From the 13 programs that generated conflicts, 10 contained simple update statements (`i++`), and 3 contained complex update statements (`i++,j++`). In both cases, Jeliot ConAn animated those statements ahead of time in each iteration of the loop until the Fault button was pressed.

9 Discussion

The evaluation and testing of the framework has provided us with useful insights about the effort required to create conflictive animations from different sources. It has also allowed us to better identify the relationship between the effort to create conflictive animations and the usage of these animations in authentic settings.

Once a new conflictive object is implemented in the framework, the framework can reliably produce a conflictive animation for that object when the conditions are met. Only in one occasion did the framework fail to produce the conflictive animation when it should have done. In this case, the interpreter-based execution could not gather the required information about the conflict. We do not consider this limitation of the framework to be significant, as for the rest of the cases, the framework worked as expected. The occurrence of the limitation is due to how the Java interpreter builds new objects using the Java virtual machine. If new conflicts related to object creation in Java were to be implemented in the framework, we would recommend building a test set in advance to be sure that the conflict object will be suitable for a wide range of programs.

These results show the potential to create conflictive animations from other textbooks or teachers' own example programs. This way, teachers can make use of conflictive animations when they think it is necessary, and do not need to be concerned about adapting their examples to run in Jeliot ConAn.

Comparatively, implementing the "for update" statement conflict was significantly faster than implementing the inheritance conflicts. This factor, combined with the fact that every program tested in the evaluation that contained a *for* loop correctly produced a conflictive animation, shows us that it is important to assess the relevance of the possible conflict in a repository of Java examples. Making conflictive animations applicable to a wide range of existing examples will make their adoption by educators easier.

The animation of inheritance conflicts only happened in half of the example programs. The implemented conflicts relate to subtle Java specification details that are not always taught in elementary Java courses, and thus, they less likely to appear in the example programs. However, the ability to create conflicts at a more advanced levels should attract experienced students to use Jeliot ConAn and benefit from the animation.

According to the effortlessness taxonomy [8], Jeliot ConAn can be regarded as a low effort tool for adoption by instructors. The *scope* of the tool can be considered course wide, as it can create animations for most programming concepts taught in CS1 or CS2. Jeliot ConAn's *integrability* lacks the possibility to customize the animation or the interaction; however, the *interaction* of Jeliot can be regarded as low effort from the producer point of view. The evaluation in the previous section has shown how with little modification programs can make use of conflictive animations. Moreover, students can also use the examples on their own. From the user point of view, several ways of interacting are possible as described in Table 4 following the engagement taxonomy [6]. Jeliot ConAn can be a tool to create activities for passive *viewing* and interactive viewing (*responding*). As it is focused on the automatic generation of animations, Jeliot ConAn cannot be used by students to manipulate the animation to create their own animations—animations can only react to changes in the source code.

We have discussed how conflictive animations have to take into account the existing body of examples and cater for both novice and advanced students. Another factor that could improve their value is by creating a set of conflictive animations that reveal common misconceptions students have when learning programming [23].

The kind of exercise implemented in Jeliot ConAn—detecting and identifying the wrong step in an animation—has not been considered in previous taxonomies of visualization exercises [17], [24]. These taxonomies consider whether the input and output are known in advance or not, but they do not explicitly mention the correctness of their values. For example, specifying the correct or incorrect content of the output in an exercise could make new kinds of exercises available. Here, we have only considered one of these exercises, where the program is correct (that is, it does not have semantic or syntactic errors, the input to the program is fixed, and the exercise question lies in the output which is a conflictive and erroneous animation). Several other exercises could be devised if we take into account the correctness of the different elements, such as an incorrect program producing correct animations.

We are currently analyzing data from an empirical evaluation of the pedagogical effect of conflictive animations compared to the use of well-behaved animations, and initial results show improvement over well-behaved animations. Benefits from the conflictive animations could be well similar to what Große and Renkl discovered: students with good prior knowledge will benefit the most when trying to look for the errors by trying to explain the error [15].

10 Conclusion

Jeliot 3's modular architecture has been extended to add support for alternate and potentially conflictive animations. Jeliot 3 has been converted to Jeliot ConAn, which can automatically create conflictive animations. Three concepts are already implemented in Jeliot ConAn for potential conflictive animation including basic loop operation and advanced inheritance concepts. Thus, any source code that contains at least one of the implemented *conflictive concepts* leads to a conflictive animation that students can interact with. The framework that has been implemented in Jeliot ConAn provides the structure to add more conflicts for other concepts with relative ease.

Jeliot 3 has resulted in a flexible platform for developing the idea of conflictive animations in a way that is almost effortless for teachers. Teachers can use their examples with few or no modifications to produce conflictive animations activities. In implementing Jeliot ConAn, two main decisions were made: using the tree interpreter layer and focusing on *expressions* and *statements* to create the conflicts. This combination has been technically a success as it has enabled us to create conflictive animations that are not trivial for students to identify and that expose possible misconceptions students might have.

In the paper, we have discussed only three conflictive objects, which relate to two programming concepts. However, we have laid out the foundation for creating new conflictive objects within the framework. Future iterations of the tool will include more conflictive objects that address other important programming topics. New conflictive objects will be evaluated according to the effort necessary to use textbook examples as sources for conflictive animations.

Appendix

List of Java resources:

- Rogers Cadenhead, *Java 2 Trainer*. ITPress, 1999.
- David Flanagan, *Java in a Nutshell*. O'Reilly, 2005.
- Pekka Kosonen, Juha Peltomäki, and Simo Silander, *Java 2 Ohjelmoinen peruskirja*. Docendo Finland Oy, 2005.
- Y. Daniel Liang, *Introduction to Java Programming*. Pearson Education, 2009.
- Sun Microsystems, The Java Tutorials, http://java.sun.com/doc/books/tutorial, retrieved on 20 Jan. 2009.
- Arto Wikla, *Ohjelmoinnin perusteet Java-kielellä*. OtaDATA, 2003.

References

[1] A. Moreno, E. Sutinen, R. Bednarik, and N. Myller, "Conflictive Animations as Engaging Learning Tools," *Proc. Seventh Baltic Sea Conf. Computing Education Research (Koli Calling '07),* R. Lister and B. Simon, eds., vol. 88, pp. 203-206, 2007.

[2] R. Ben-Bassat Levy, M. Ben-Ari, and P.A. Uronen, "The Jeliot 2000 Program Animation System," *Computers and Education,* vol. 40, no. 1, pp. 1-15, 2003.

[3] A. Moreno and M.S. Joy, "Jeliot 3 in a Demanding Educational Setting," *Electronic Notes in Theoretical Computer Science,* vol. 178, pp. 51-59, 2007.

[4] J. Sorva, "The Same But Different: Students' Understandings of Primitive and Object Variables," *Proc. Eighth Baltic Sea Conf. Computing Education Research (Koli Calling '07),* A. Pears and L. Malmi, eds., 2008.

[5] C.D. Hundhausen, S.A. Douglas, and J.T. Stasko, "A Meta-Study of Algorithm Visualization Effectiveness," *J. Visual Languages and Computing,* vol. 13, no. 3, pp. 259-290, 2002.

[6] T.L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, and J.Á. Velázquez Iturbide, "Exploring the Role of Visualization and Engagement in Computer Science Education," *Proc. Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '02),* pp. 131-152, 2002.

[7] T. Naps, S. Cooper, B. Koldehofe, C. Leska, G. Rößling, W. Dann, A. Korhonen, L. Malmi, J. Rantakokko, R.J. Ross, J. Anderson, R. Fleischer, M. Kuittinen, and M. McNally, "Evaluating the Educational Impact of Visualization," *Proc. Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '03),* pp. 124-136, 2003.

[8] P. Ihantola, V. Karavirta, A. Korhonen, and J. Nikander, "Taxonomy of Effortless Creation of Algorithm Visualizations," *Proc. First Int'l Workshop Computing Education Research (ICER '05),* pp. 123-133, 2005.

[9] N. Postman, "The Fallen Angel," *The End of Education,* Vintage Books, 1996.

[10] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari, "Visualizing Program with Jeliot 3," *Proc. Int'l Working Conf. Advanced Visual Interfaces (AVI '04),* pp. 373-380, 2004.

[11] A. Kumar and S. Kasabov, "Observer Architecture of Program Visualization," *Electronic Notes in Theoretical Computer Science,* vol. 178, pp. 153-160, 2007.

[12] J. Ángel Velázquez Iturbide and C. Pareja-Flores, "An Approach to Effortless Construction of Program Animations," *Computers & Education,* vol. 50, no. 1, pp. 179-192, 2008.

[13] V. Karavirta, A. Korhonen, L. Malmi, and K. Stalnacke, "MatrixPro—A Tool for Demonstrating Data Structures and Algorithms Ex Tempore," *Proc. IEEE Int'l Conf. Advanced Learning Technologies (ICALT '04),* pp. 892-893, 2004.

[14] M. Ben-Ari, "Constructivism in Computer Science Education," *J. Computers in Math. and Science Teaching,* vol. 20, no. 1, pp. 45-73, 2001.

[15] C. Große and A. Renkl, "Finding and Fixing Errors in Worked Examples: Can This Foster Learning Outcomes?" *Learning and Instruction,* vol. 17, pp. 612-634, 2007.

[16] A. Rudder, M. Bernard, and S. Mohammed, "Teaching Programming Using Visualization," *Proc. Sixth Conf. IASTED Int'l Conf. Web-Based Education (WBED '07),* pp. 487-492, 2007.

[17] A. Korhonen and L. Malmi, "Taxonomy of Visual Algorithm Simulation Exercises," *Proc. Third Program Visualization Workshop,* pp. 118-125, 2004.

[18] L. Ma, J.D. Ferguson, M. Roper, I. Ross, and M. Wood, "Using Cognitive Conflict and Visualisation to Improve Mental Models Held by Novice Programmers," *Proc. 39th SIGCSE Technical Symp. Computer Science Education (SIGCSE '08),* pp. 342-346, 2008.
[19] N. Myller, "The Fundamental Design Issues of Jeliot 3," master's thesis, Univ. of Joensuu, 2004.
[20] A. Moreno, "Intermediate Code in Program Animation Software," master's thesis, Univ. of Joensuu, 2005.
[21] A. Moreno, E. Sutinen, and M. Joy, "Understanding and Evaluating Conflictive Animations," to be submitted.
[22] N. Ragonis and M. Ben-Ari, "Teaching Constructors: A Difficult Multiple Choice," *Proc. 16th European Conf. Object-Oriented Programming Workshop,* vol. 3, 2002.
[23] S. Holland, R. Griffiths, and M. Woodman, "Avoiding Object Misconceptions," *Proc. 28th SIGCSE Technical Symp. Computer Science Education (SIGCSE '97),* pp. 131-134, 1997.
[24] M. Bruce-Lockhart, P. Crescenzi, and T. Norvella, "Integrating Test Generation Functionality into the Teaching Machine Environment," *Proc. Fifth Program Visualization Workshop*, G. Rößling and J.Á. Velázquez Iturbide, eds., vol.. 224, pp. 115-124, 2009.

Andrés Moreno is a PhD student in the Department of Computer Science and Statistics at the University of Joensuu. He is currently working in educational technology and his main interests are visualization tools, programming education, and ICT for development. He has been an active developer of the programming visualization tool Jeliot 3, which he has used to teach programming in Finland and Tanzania, and he has published more than 20 papers in journals and conferences.

Mike Joy received the masters' degrees in mathematics from Cambridge University and in postcompulsory education from the University of Warwick, the PhD degree in computer science from the University of East Anglia, and has also received both the CEng and CSci degrees. He is an associate professor in the Department of Computer Science at the University of Warwick and a member of the Intelligent and Adaptive Systems research group. His research interests include educational technology, computer science education, object-oriented programming, and Internet software, and he is the author or coauthor of more than 100 papers. He is a chartered fellow of the British Computer Society and a fellow of the Higher Education Academy.

Niko Myller received the BSc degree in 2003, the MSc degree in 2004, and the PhD degree in 2009, from the Department of Computer Science and Statistics at the University of Joensuu. His research interests lie in the fields of visualization and concretization technologies, CSCL, information retrieval, computer ethics as well as adaptive systems. He has published more than 40 papers related to these topics in international journals and conferences.

Erkki Sutinen received the PhD degree in computer science from the University of Helsinki in 1998. He is the leader of the edTech group (www.cs.joensuu.fi/edtech). Since 2006, he has been the head of the Department of Computer Science and Statistics at the University of Joensuu. His research interests include ICT for development (ICT4D) and designing and analyzing technologies for complex subject domains, like programming, in developing countries, and within special education. The applied techniques cover visualization, information retrieval, data mining, robotics, and design models. He has coauthored and published more than 100 research papers, and 12 of his supervised or cosupervised PhD students have completed their studies. He has also worked at Purdue University (1998-1999), the University of Linköping (2000-2001), and Massey University (2006), and is an adjunct professor at Tumaini University, Tanzania.