Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 32 (2014) 561 – 570
Procedia Computer Science

5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014)

Undo/Redo Operations in Complex Environments

Karel Jakubec, Marek Polák*, Martin Nečaský, Irena Holubová

Department of Software Engineering – Charles University in Prague, Malostranské nám. 25, Prague 118 00, Czech Republic

Abstract

During the past thirty years, several types of non-linear undo models have been presented, but almost none of them solves undoing and redoing actions in environments, where multiple history buffers are involved and when there are causal dependencies among separate actions.

This paper describes a new model which allows a user to select any action from any history buffer. The key part of the model is a smart command design and an undo manager, which searches for dependencies and offers possible solutions to the user. The results are presented in the context of evolution-management framework called *DaemonX*.

Keywords: information systems, undo/redo operations, evolution management

1. Introduction

An undo/redo functionality is today a common feature of various interactive applications and systems. Most of today's applications use a very simple model – all user's actions are stored in a history buffer (stack). Only the most recent action may be undone by performing an inverse operation. Such an action is marked as undone and the user may redo it or undo the previous action. This simple model, commonly known as *linear undo model*, is sufficient in many applications. Its simplicity is a big advantage – most of the users are familiar with it and they expect such behavior. The result of undo/redo operation may be easily predicted and also the physical implementation is not very complex using the *Command design pattern*[1] which divides user's actions to a series of discrete steps, which can be later undone.

But there are environments, where this approach may not be sufficient[2]. If there are non-trivial dependencies among actions – for instance when one document is being edited simultaneously in multiple workspaces or one object is being modified from different perspectives – the simple linear undo model may not be user-friendly. Hence, the respective task is to implement a model, which allows a user to undo any action at any time – so-called *selective undo*. This feature can greatly save user's time, but its results tend to be unpredictable and it is hard to implement even in simple environments.

* Marek Polák. Tel.: +420-221-914-260; fax:+420-221-914-260.
E-mail address: polak@ksi.mff.cuni.cz

doi:10.1016/j.procs.2014.05.461

The aim of this paper is to explore the possibilities of undo/redo management in an indicated complex environment and create an algorithm which allows the user to undo any command at any time at any place in the environment. The proposed algorithm deals with documents spread over several workspaces and successfully manage correct undoing of actions, which are dependent on each other. The algorithms were implemented in *DaemonX*[3] – a framework for data modeling and evolution management developed at the Faculty of Mathematics and Physics of the Charles University in Prague. Its main purpose is to provide a set of tools to model a modeling language (also known as *meta-modeling*), a mechanism for data propagation among various modeling languages and a common runtime environment. One document in *DaemonX* may consists of several diagrams, each in a different modeling language. This fact has brought unexpected difficulties during implementation of undo/redo functionality, because the linear undo model was slow and the results were not sufficient for effective work with the framework.

The paper is structured as follows: In Section 2 we introduce the undo/redo functionality in general and in Section 3 we describe the existing undo models. In Section 4 we introduce our novel undo/redo algorithms and in Section 5 we describe their implementation. In Section 6 we present the results of using the presented model. We conclude in Section 7.

2. Undo/Redo Functionality

For the purpose of this paper we suppose, that the user works with objects called *constructs*. A construct represents a general object, e.g. a UML class in a class diagram. Each has a unique identifier (*id*), a set of *properties* and *views*. The properties may be considered as a simple `(key, value)` pair. The view is a visualization of the construct to the user. One construct may have several views, each showing the construct from a different perspective. In many situations, one view is sufficient, but there are situations, when a different view of the properties may be useful (e.g. an overview and a detail). The view is usually used for changing the data properties of the construct. This principle is more deeply described by the design pattern *Model-View-Controller*[1].

All constructs created during user's work reside in a data structure called *construct pool*. This data structure can be implemented in various ways (e.g. array, linked list, hashmap,...). A system may also have more than one construct pool. An application may also have several *workspaces*. A workspace is a place, where the user performs his/her modifications of constructs. For example, in a designer of diagrams, each diagram can be a separate workspace. Several workspaces may share one construct pool.

2.1. Commands

In an editor there must exist a mechanism how to delimit and pick exactly one user action – the user's work should be a series of discreet steps. These steps are called *commands*. Each command is a separate unit of action which can be either executed or undone and stored for later usage.

A command is indivisible for the user. If the user wants to undo a command, (s)he must undo it as a whole. But from the application perspective the command does not have to be indivisible. For this reason we introduce so-called *atomic commands*. An atomic command is an atomic action from the application point of view and typically it is used to create, delete or modify one construct or its property. *Composite commands*, used by users, may then serve as containers for atomic commands. We assume that two commands cannot interleave and cannot be performed at the same time.

The ordering of commands enables to distinguish in which order commands should be undone or redone. Ordering can be *local* (only one history buffer is involved, often implicit by command's position on the stack) or *global* (among all commands in the application).

Definition 1. *Command C_1 is* older *than command C_2 if and only if C_1 is a predecessor of C_2, i.e. C_1 was originally executed before C_2 was executed. Command C_1 is* younger *than command C_2 if and only if it is not older than C_2.*

Since equality is not defined, it is always possible to determine, which command is older/younger.

2.2. History Logging

There are generally two ways in which the command history may be logged:

1. The history is logged as a series of document states and a model describes the way how to move from one state to another.[4]
2. The history is logged as a series of operations (similar to commands) and undo is performed as appending an inverse operation to the history buffer.[5]

The former approach is memory consuming (each state of the document is saved) and since there is generally no system of dependencies among various states, it is harder to support more advanced features of undo models like the selective undo. For this reason this paper deals with the second choice – the executed commands are stored in a history buffer, whose implementation depends on the used undo model, but in most of today's applications it is a simple *LIFO* (First In First Out) container.

2.3. Dependencies among Commands

In very simple environments, commands are usually independent of each other. These environments typically do not allow the user to modify created constructs, and it is not allowed to derive one construct from another or to build up relations among various constructs (for example see thesis[6]). However, generally, there are two types of dependencies:

1. *Implicit dependency* : If command A modifies construct C_1 and a younger command B also modifies C_1, the result of command B is dependent on the result of command A. We say, that command B *depends on* command A. If command B depends on command A and command C depends on B, than command C depends also on A – i.e. the *transitivity* holds.
2. *Explicit dependency*: This dependency is set by the system. It complements implicit dependencies in cases when one command uses the result of another command, but they do not work with the same construct. The transitivity also holds.

Implicit dependencies do not have to be stored, they can be computed on the fly; however, it is necessary to extend the commands with information about affected construct(s). Conversely, information about explicit dependencies has to be held in a dedicated data structure. However, for undo/redo purposes it is not important, whether two commands are dependent explicitly or implicitly – the meaning is the same. Hence, we will just consider the situation when command A *depends* on command B.

3. Undo Model

An *undo model* represents the way an application approaches the undo/redo functionality. Particular models can differ from each other in many ways, but usually they have the following common parts:

- **Commands** represent an action of the user.
- **History buffer(s)** store executed commands.
- **Undo/redo manager** controls history buffers.
- **User interface** interacts with the user.

These parts can be easily mapped to the Model-View-Controller pattern[1]. Commands and history buffers form the model, the user interface represents the view and the controller is represented by the undo/redo manager. All parts of the model are usually tightly coupled, but the key part of the model is the undo/redo manager – it manipulates with stacks and reacts on user's actions by selecting commands for execution and undo. Properties of an undo model define what behaviors each model satisfies or not. We usually consider the following ones:

- **Stable Execution Property**: A command is always redone in the same state that it was originally executed in and is always undone in the state that was reached after the original execution. A state in this context is an ordered list of commands that are done.
- **Weakened Stable Execution Property**: During redo of command C, all commands, on which command C depends, are redone prior to C. During undo of command C, all commands dependent on C are undone prior to C.

- **Stable Result Property**: A command is always redone from the state in which it was originally executed and it is always undone from the state that was reached after the original execution. A *state* in this context is an ordered list of commands that were executed.
- **Commutative Undo Property**: The undo model is *commutative* if the state reached after undo/redo of any two commands C_1 and C_2 is equal to the state reached after undo/redo of C_1 and C_2 in the opposite order. An example is depicted in Figure 1.
- **Minimalistic Undo Property**: The undo model is called *minimalistic* if redo operation of command C redoes only command C and all commands older than C on which C depends, and if undo operation of command C undoes only command C and all commands younger than C which are dependent on C. An example is depicted in Figure 2.

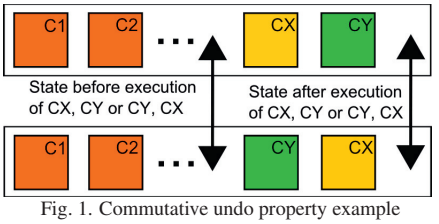

Fig. 1. Commutative undo property example

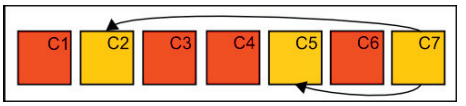

Fig. 2. Minimalistic undo property. Redo of command *C7* will execute redo of related commands *C2* and *C5*.

3.1. Existing Solutions

During the past years, several undo models were presented. This section will focus on models, which use commands as the basic unit on which undo/redo is performed.

3.1.1. Linear Undo/Redo

The *linear undo model* is the simplest (and probably the most widely used) way how to achieve the undo/redo functionality. The word linear means, that only the most recently executed command can be undone. All executed commands are stored in one history buffer – a simple LIFO container, often called *stack*. A new command is always pushed to the top of the stack. There are multiple possibilities how it can be implemented (e.g. a two stack version or a pointer version)[6].

3.1.2. Non-linear Undo and Redo

A *non-linear undo* brings one fundamental feature – there is a possibility of undoing or redoing also other commands than the last executed one. That is, any command can be undone at any time. Several models have been presented, such as, e.g., the Script model[7], the U&S model[8], Triadic model[9] or the Direct selective model[2].

3.1.3. Selective Undo Redo

Selective undo is not a model, but it is a feature which a model can offer. There is no clear definition of what selective undo is. Most papers focus on defining the correct result of the undo/redo operation. But the definition of selective undo is different, because it is not just another property which holds true or not. The important question in this case is not "What should be the correct result of undo/redo operation?", but "What functionality should the model offer to the user?".

For the purpose of this paper, we have selected fundamental functions a user should be allowed to do if we want to say that a particular model supports selective undo:

1. The user should be allowed to undo any executed action in the history buffer. Actions independent of the action being undone should be left untouched.
2. The user should be allowed to redo any undone action in the history buffer. Actions independent of the action being redone should be left untouched.
3. No command can be automatically discarded from history buffer without direct user's request.

These requirements are common and the definition does not specify what should be the correct result of a selectively undone/redone command. There are several ways how to achieve this functionality. An important aspect is that the user knows how it will be performed.

Almost all models for selective undo need to solve several issues, which all result from one simple fact: Commands may be undone or redone outside the context, in which they were originally executed – they break the stable result property. This fact is not a problem as long as there are no dependencies among commands. Depending on the application, it is also possible that the commands may be shuffled and redone in any order. There are, however, three main general issues:

- **Dead references**: Commands may be undone in significantly different contexts than they were originally executed. Constructs which are being modified by these commands may be in different states or may not even exist.
- **Modify already modified**: Let us have a construct A with property P of type `string`. After creation of A, P = "`abcd`". Then two commands C_1 and C_2 are invoked. First, C_1 sets P to "`efgh`", then C_2 to "`ijkl`". Now consider the selective undo of C_1. The question is, what is then the correct value of P.
- **Discard commands problem**: In the linear undo model, commands above the stack top are usually discarded from the stack after a new command has been executed. The selective undo requirement 3 specifies that no commands should be discarded from the stack. It is necessary because selective undo models usually do not have a pointer to the top of the history buffer and, therefore, the model cannot determine which commands should be discarded. However when the user undoes a command, his/her intention is to undo the effect of the command. The effect of the command is probably unwanted by the user and it will probably not be redone again (redo is often used just to correct a badly selected undo operation). If this logic is correct, the history stacks can be filled with many undone commands, which will never be redone, because their effects are useless for the user.

4. Proposed Algorithms

In the *DaemonX* framework[3], the highest-level unit of environment is a *document*, also called *project*. It can be considered as a container, which creates an envelope around all other objects. A document is a standalone unit. If a system supports simultaneous work with multiple documents, they cannot interfere with each other. A *workspace* is an interface for the user, which (s)he can use to modify the content of the document. One document may be modified from several workspaces and also one construct may be viewed in several workspaces. The situation is depicted in Figure 3.

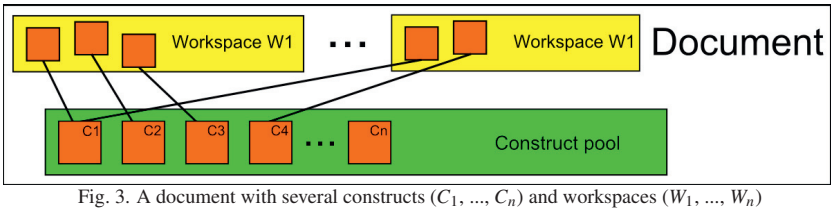

Fig. 3. A document with several constructs (C_1, ..., C_n) and workspaces (W_1, ..., W_n)

Due to space limitations, the full description of all algorithms with theirs proofs is omitted – it can be found in[6], namely the first two models, i.e. the *Extended Linear Undo* and the *Cascade Selective Undo*. In the following text we will describe their combination and improvement called the *Combined Undo*.

4.1. Combined Undo Model Approach

The key idea of the *Extended Linear Undo* is to take a plain linear undo model and modify it in the way, that it suits for environment with several workspaces. To satisfy intuitiveness and easy implementation for existing applications, it would be useful, if each workspace would manage its own history buffer – if a command is created in the workspace, it goes to the top of its buffer. This approach results from one assumption: If a user is working in particular workspace,

he/she wants to undo or redo commands executed in this workspace prior to commands in other workspaces. Because there can be relations between commands in different stacks, to satisfy correct undo the exact order of commands among stacks must be known – it is achieved by sequence numbering of each command.

In *Cascade Selective Undo* the command can be undone or redone in the middle of the stack without undo/redo of commands which are not dependent on the command being undone/redone – they can be left untouched. To provide correct undo/redo operation of the command, all dependent commands must be undone/redone prior to the selected command and therefore the dependency search function must be run to construct collection of dependent commands before actual undo or redo is performed.

Combined Undo Model is to take the best from both models – the intuitiveness of the Extended Linear Undo model and the possibilities that the Cascade Selective Undo offers.

The Cascade Selective Undo model brings the possibility of undoing any command at any time. But there is one situation which it handles badly – when the user wants to get the whole document (all the workspaces) to some specific state (so called *global linear undo*), which once existed in the past after execution of a particular command. In this case the user has to undo each command manually which is not much user-friendly if the requested command is very old and, thus, deeply nested.

The Extended Linear Undo model behaves better in this situation, but the result also depends on the number of dependencies among commands. The advantage is that the user can usually select a command in the middle of the buffer and all the commands above it are undone. Bringing the document to a specific state means at least one click in each history buffer. This is significantly better in comparison with the Cascade Selective Undo model, but the optimal solution would be just one special undo action. Another problem is that the user has to know the global order of commands in all stacks.

The properties to be satisfied are as follows: **Weakened stable execution property**, **Stable result property**, **Commutative undo property** and **Minimalistic undo property**. The stable execution property cannot be true, if we want to offer any command to undo/redo and simultaneously obey the minimalistic undo property. We would also like to offer a possibility of performing undo in the traditional way (undo/redo button), which implies implementation of a mechanism which will select one command to undo/redo in each stack. The behavior of the model is as follows:

1. Each executed command can be undone.
2. Each undone command can be redone.
3. The youngest executed command is offered for linear undo. When such a command does not exist, the linear undo is not possible.
4. If there is a continuous sequence of undone commands at the end of the history buffer, the oldest command from this sequence is offered for linear redo.
5. Each executed command can be selected for global linear undo. After performing global linear undo on command C_{toUndo}, there is no younger executed command than C_{toUndo} and no older undone command than C_{toUndo} in any history buffer in the document.
6. After a successful call of execute operation on command $C_{toExecute}$ in workspace W, $C_{toExecute}$ is executed and it becomes the last command in the history buffer belonging to W.
7. After a successful call of undo operation on command C_{toUndo} in workspace W, C_{toUndo} is undone.
8. After a successful call of redo operation on command C_{toRedo} in workspace W, C_{toRedo} is redone.
9. Commands are never discarded from the history buffer.

4.1.1. Principle and Analysis

The combined undo model is a mix of the Extended Linear Undo model and Cascade Selective Undo model, which are both described and proven to be correct. As mentioned before, if we want to allow a model to act like the linear undo model, there must be a mechanism which selects which commands will be undone or redone when the undo or the redo button is pressed. Extended linear undo model uses a pointer to the top of the stack, where the border between undone and executed commands lies. But if we support selective undo and, therefore, commands in the middle of the stack may be undone, there can be more than one such border. For this purpose, we create a pointer to the so-called *virtual stack top*. This pointer always points one position above the youngest executed command. It can be easily computed on the fly by iterating through history buffers. The command to undo is the one bellow the virtual stack top, the command to redo is the one at which the pointer points.

Global linear undo is a new feature, which is neither present in the Extended Linear Undo model, nor in Cascade Selective Undo model. It is called "undo", but it may also perform redo operations if necessary – all undone commands older than the selected command have to be redone and all executed commands younger than the selected command have to be undone. Its implementation can be easily done by one iteration through the *global history buffer*, a simple collection for command ordering according to their sequence number.

It should be noted, that the state reached after performing global linear undo may not be the exact state, in which the document was after the original execution of the selected command. If there were any undone commands present in any stack during the original execution, the global linear undo will redo these commands. Saving the exact state of stacks after execution of each command cannot be done – it would be a new linear undo model (based on saving document state after each step).

Execution of a new command would be done in the way that selective undo requires – commands to redo are not discarded. It would be possible to discard all commands above the virtual stack, but it would collide with definition of the selective undo.

4.1.2. Data Structures and Algorithms

The `Command` structure and the `History buffer` structure are depicted in Figure 4. As the pointer to the top of the stack will be computed on the fly, we do not need a variable to store this information, which makes this implementation sufficient. But the functions which manipulate with the history buffer are changed, because we would like to support also the linear undo.

```
struct {
    int sequence_number;
    int keys_of_affected_constructs[];
    bool is_undone;
    void* user_data;
} Command;
```

```
struct {
    Command command_stack[];
} History_buffer;
```

Fig. 4. The basic data structures

The list of basic functions used in the algorithm is as follows:

- `ItemCount(General_collection)` – Returns the number of items in a general collection.
- `Append(General_collection, Item)` – Adds one item at the end of the collection.
- `GetCommandByPosition(General_collection, Index)` – Returns the item which resides on the `Index`'th position of the collection. If there is no such item, the null value is returned. `Index` is zero based.
- `GetCommandByKey(General_collection, Key)` – Returns the item with the specified `Key` (in case of a command with specified `sequence_number`). If there is no such item, null value is returned.
- `GetVirtualStackTop(General_collection)` – Returns the index of the virtual top of the stack.
- `GetCommandToLinearUndo(History_buffer)` – Returns the command, which is right under the virtual top of the stack or null value, if such a command does not exist.
- `GetCommandToLinearRedo(History_buffer)` – Returns the command at which points the virtual top of the stack or the null value, if such a command does not exist.

The global history buffer serves only as a simple collection for command ordering according to their sequence number.

The algorithm has to support three different types of undo/redo:

- **Linear** – Acts in the same way as the Extended Linear Undo model.
- **Selective** – Acts in the same way as Cascade Selective Undo model.
- **Global** – Gets the document into a state in which it once was after the original execution of the selected command.

The functions for dependency search (`FindDependentCommandForUndo` and `FindDependentCommandForRedo`) are parameterized by one Boolean variable `is_linear`, which specifies, whether the function searches for the dependencies as the Extended Linear Undo model requires (if true) or as the Cascade Selective Undo model requires (if

false). Both return a collection of commands, which should be reversed and then undo or redo may be performed on each member of the collection.

The next important function is `Execute` (Algorithm 1), which executes a new command and appends it to the history buffer.

There are together four functions for undo and redo – `UndoCommandSelective` (Algorithm 2), `RedoCommandSelective` (Algorithm 4), `UndoCommandLinear` (Algorithm 3) and `RedoCommandLinear` (Algorithm 5). Both selective and linear functions take different number of arguments, so they are really presented as four distinct functions.

The last function which allows the user to undo and redo commands is called `UndoCommandGlobal` (Algorithm 6). The purpose of this function is to get the document into a state, when there are no younger executed and no older undone commands in the whole document than selected command.

Algorithm 1 Execute command in combined undo model

```
1: procedure ExecuteCommand(workspace_key, command)
2:     history_buffer ← GetHistoryBuffer(workspace_key)
3:     if Execute(command) then
4:         Append(history_buffer, command)
5:         Append(global_history_buffer, command)
6:         return true
7:     end if
8:     return false
9: end procedure
```

Algorithm 2 Selective undo command in combined undo model

```
1: procedure UndoCommandSelective(workspace_key, com_index)
2:     com2undo ← GetCommandByIndex(workspace_key, com_index)
3:     // Command to undo
4:     if com2undo ≡ NULL then
5:         return false
6:     end if
7:     com2undo_collection ← FindDependentCommandsForUndo(com2undo,
   false)
8:     // Collection of commands to undo
9:     Revert(com2undo_collection)
10:    for i ← 0 to ItemsCount(com2undo_collection) do
11:        Undo(com2undo_collection[i])
12:    end for
13:    return true
14: end procedure
```

Algorithm 3 Linear undo command in combined undo model

```
1: procedure UndoCommandLinear(workspace_key)
2:     com2undo ← GetCommandToUndo(workspace_key)
3:     // Command to undo
4:     if com2undo ≡ NULL then
5:         return false
6:     end if
7:     com2undo_collection ← FindDependentCommandsForUndo(com2undo,
   true)
8:     // Collection of commands to undo
9:     Revert(com2undo_collection)
10:    for i ← 0 to ItemsCount(com2undo_collection) do
11:        Undo(com2undo_collection[i])
12:    end for
13:    return true
14: end procedure
```

Algorithm 4 Selective redo command in combined undo model

```
1: procedure RedoCommandSelective(workspace_key, com_index)
2:     com2redo ← GetCommandByIndex(workspace_key, com_index)
3:     // Command to redo
4:     if com2redo ≡ NULL then
5:         return false
6:     end if
7:     com2redo_collection ← FindDependentCommandsForRedo(com2redo,
   false)
8:     // Collection of commands to undo
9:     Revert(com2redo_collection)
10:    for i ← 0 to ItemsCount(com2redo_collection) do
11:        Redo(com2redo_collection[i])
12:    end for
13:    return true
14: end procedure
```

Global history buffer is a perfect structure for the implementation of such a function. The basic idea is an iteration through the buffer, which build up two collections – commands to redo and commands to undo. Each undone younger command goes to the commands to redo, each older executed command to commands to undo. No other dependencies have to be searched, because if there are some dependent commands to undo or redo, they will be surely also undone or redone.

Algorithm 5 Linear redo of command in combined undo model

```
1: procedure RedoCommandLinear(workspace_key)
2:     com2redo ← GetCommandToRedo(workspace)
3:     // Command to redo
4:     if com2redo ≡ NULL then
5:         return false
6:     end if
7:     com2redo_collection ← FindDependentCommandsForRedo(com2redo,
       true)
8:     // Collection of commands to redo
9:     Revert(com2redo_collection)
10:    for i ← 0 to ItemsCount(com2redo_collection) do
11:        Redo(com2redo_collection[i])
12:    end for
13:    return true
14: end procedure
```

Algorithm 6 Global linear undo

```
1: procedure UndoCommandGlobal(workspace_key, com_index)
2:     selected_comm ← GetCommandByIndex (workspace_key, com_index)
3:     if selected_com ≡ NULL then
4:         return false
5:     end if
6:     i ← GetIndexToGlobalBuffer (selected_comm.sequence_number)
7:     comm2redo ← empty_collection
8:     for i ← 0 to index – 1 do
9:         comm ← global_command_buffer[i]
10:        if comm.is_undone then
11:            comm2redo ← comm2redo ∪ comm
12:        else
13:            continue
14:        end if
15:    end for
16:    comm2undo ← empty_collection
17:    for i ← ItemCount(global_command_buffer) downto index do
18:        comm ← global_command_buffer[i]
19:        if comm.is_undone then State comm2undo ← comm2undo ∪
       comm
20:        else
21:            continue
22:        end if
23:    end for
24:    for i ← 0 to ItemsCount(comm2redo) do
25:        Redo(com2redo[i])
26:    end for
27:    for i ← 0 to ItemsCount(comm2undo) do
28:        Undo(com2undo[i])
29:    end for
       State return true
30: end procedure
```

Two final for loops can be switched, because in case of redo, we start with the oldest undone command (which implies that no older dependent undone command exists) and in case of undo we start with the youngest executed command (which similarly implies that no younger dependent executed command exists).

5. Implementation

Project *DaemonX*[3] has been chosen to serve as a platform for experiments with the proposed undo models. The *DaemonX* framework is plug-in-able tool developed for data and/or process modeling and evolution management framework. All functionality is provided via various plug-ins (mainly modeling and evolution) which use services provided by *DaemonX* framework. The main services provided by *DaemonX* are as follows: integrated environment, support of of plug-ins and their inter-operability, propagation of changes among models created by plugins. The implementation details are omitted for space limitation.

6. Results

All models were tested during the development and usage of *DaemonX* by developers and users. They gave us feedback to be able to improve the algorithm and behavior to be more intuitive and useful in usage of the tool while designing complex multilayered models. The ability of the proposed algorithms was compared with other similar solutions like Visual Studio[10] and Enterprise Architect[11]. The Combined Undo model algorithm offers more complex undo/redo functionality than the mentioned tools even though both of them work with complex models.

Since the framework now supports three methods for undo, the user has to be able to select which one (s)he wants to use. The command stack interface was extended with the context menu, which reacts on right mouse button (RMB). When the user clicks RMB on the command (s)he can select from the menu, whether (s)he wants to use linear undo/redo, selective undo/redo or global linear undo. Selecting linear undo (redo) on the command in the middle of the stack causes linear undo (redo) on all executed (undone) commands above (below). Buttons “undo” and “redo” are present and they cause linear undo of the command right below or at the virtual stack top.

The combined undo model takes the best from both previous models. When the user wants to undo only a single command in the middle of the stack, (s)he can use a selective undo. If there is a need to undo a big chunk of commands, linear undo or global linear undo can be used. The global linear undo feature was really appreciated by the users which shows that linear approach is still probably more natural than selective. The user interface – undo/redo buttons and the list of commands with context menu – is generally sufficient, but its improvement can be a part of possible future work.

7. Conclusion

The aim of this paper was to propose a new undo/redo approach that would suit the needs of a complex environment with multiple workspaces. Three undo models were proposed and their advantages and disadvantages were discussed. Their proof of concept was provided within the system *DaemonX*[3], a general modeling and evolution management framework which enables to model components of an application in various types of modeling languages. All the three models are generally easy to implement, because they are built upon a common system of commands and history buffers which are widely used in today's applications.

The general aim of the paper has been fulfilled, but naturally not all issues of undo/redo operation are solved. A big challenge is the design of a selective undo model, which would search for dependencies among commands based on a kind of multi-criteria basis. The presented models use only keys of affected constructs to decide, whether there is a dependency between two commands. Future models may use also other criteria – for instance the type of command (creation, modification, deletion, move, etc.) can be taken into account. This approach could reduce the amount of commands selected as dependent in one undo/redo operation. It implies revisiting the *modify already modified* issue by careful selecting criteria which make the command dependent (e.g. movement commands depend on creation, but not on modification etc.).

Acknowledgment

Supported by the project SVV-2014-260100 and the GAUK grant no. 1416213.

References

1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Professional, 1995.
2. T. Berlage, "A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects," *ACM Trans. Comput.-Hum. Interact.*, vol. 1, pp. 269–294, September 1994.
3. K. Jakubec, M. Polak, V. Kudelas, M. Chytil, and P. Pijak, *DaemonX Framework*, 2013, http://daemonx.codeplex.com/.
4. G. B. Leeman, Jr., "A Formal Approach to Undo Operations in Programming Languages," *ACM Trans. Program. Lang. Syst.*, vol. 8, pp. 50–87, January 1986.
5. C. Sun, "Undo as Concurrent Inverse in Group Editors," *ACM Trans. Comput.-Hum. Interact.*, vol. 9, pp. 309–361, December 2002.
6. K. Jakubec, "Management of Undo/Redo Operations in Complex Environments," 2012, http://www.ksi.mff.cuni.cz/~holubova/dp/Jakubec.pdf.
7. J. E. Archer, Jr., R. Conway, and F. B. Schneider, "User Recovery and Reversal in Interactive Systems," *ACM Trans. Program. Lang. Syst.*, vol. 6, pp. 1–19, January 1984.
8. J. S. Vitter, "US&R: A New Framework for Redoing (Extended Abstract)," *SIGSOFT Softw. Eng. Notes*, vol. 9, pp. 168–176, April 1984.
9. Y. Yang, "Undo Support Models," *International Journal of Man-Machine Studies*, vol. 28, no. 5, pp. 457–481, 1988.
10. (2014) Visual Studio. Microsoft. [Online]. Available: http://www.visualstudio.com/
11. (2014) Enterprise Architect. Sparx Systems. [Online]. Available: http://www.sparxsystems.com/products/ea/index.html