Editable Replay of IDE-based Repetitive Tasks

Abstract

Programmers often have to do many repetitive tasks when using an IDE (Integrated Development Environment). These tasks require them to navigate through many views and dialogs in the same steps and input same data, which are time consuming and boring. In this paper, we present an approach to automatically perform the repetitive tasks by catching user actions on the IDE and replaying them when necessary. The sequence and contents of the caught user actions can be edited for generating user actions of similar tasks. The user actions are manifested as a set of high-level information so that they are easy to be edited and robust to UI changes. We present SmartReplayer, an implementation of our approach in the Eclipse IDE and use examples to show that it can greatly improve efficiency of Eclipse Programmers.

1. Introduction

Programmers often have to do many repetitive tasks when using an IDE (Integrated Development Environment). Such as checking out files from a CVS repository, backing up them, reformatting source codes, configuring programming contexts, etc.

Generally speaking, each of these tasks is executed as a sequence of user actions on the IDE. A user action is an operation step programmers performed via input devices such as the mouse and keyboard to operate the IDE. Performing repetitive tasks can be categorized into two types: one is to redo the same operations all the time; the other allows programmers to do some changes during the process, take the task of checking out a CVS project as an example, programmers may choose to check out a different version of the project when repeating this task. A programmer can perform a series of repetitive tasks specific for himself/herself; a team of programmers can also have series of repetitive tasks common to this group.

Doing repetitive tasks is time consuming and boring. Programmers often have to navigate through many views and dialogs by following the fixed steps one by one to access the needed information. Obviously, doing repetitive tasks automatically is a natural solution to relieve programmers of the burden. In general, there are three ways to automate operations on an IDE, the wizard, macro and visual replay:

• A wizard is a UI component which can guide programmers through several operation steps. Programmers can build different wizards for an IDE to handle different repetitive tasks. But building a wizard is not an easy work. Additionally, as there are many different kinds of repetitive tasks to a single programmer, let alone a team of programmers, the number of wizards needed to be built will be huge.

• A macro is a piece of software that allows a user to record mouse and keyboard functions for playback at a later time. But the recorded information is often low-level mouse and keyboard events which brings two limits: one is not robust enough to UI changes so that some human operations which change UIs will make the playback fail; the other is difficult to understand so that a macro is hard to be edited for generating macros of similar tasks.

• The visual replay is a mechanism that represents the Human-IDE interactions visually. It records the information derived from the IDE and the context when programmers perform some tasks, then uses the recorded information to mimic the human operations on the IDE for replaying the corresponding tasks. The challenges are to record the proper information and translate them to Human-IDE interactions when necessary.

We argue that the visual replay is a promising technique to automate the repetitive tasks, better than the wizard and macro. It can use the recorded information to help improve the current task. Additionally, through visually representing tasks, it can help new programmers to learn the task-execution processes vividly. The recorded information can also be analyzed and edited to meet different needs. In this paper, we present an editable visual replay approach to help programmers to do the IDE-based repetitive tasks automatically. The recorded information in our approach is the user actions when programmers performing tasks. These user actions can be replayed to repeat the corresponding tasks without human intervention. A user action consists of high-level information so that it can be more robust to UI changes when replayed. The recorded user actions can be represented as an action-tree for editing, such as deleting an action, changing the input parameters of an action, tagging an action, etc. We also design some algorithms to ensure that the modified sequence of user actions can be replayed correctly. We implemented our approach as a plugin in Eclipse, called SmartReplayer, so that repetitive tasks when using Eclipse can be automatically done.

The rest of this paper is organized as follows: we begin with an illustrative example in section 2; then describe how user actions are captured and replayed in section 3; we go on to present how our approach allows programmers to edit these recorded user actions in section 4; then present evaluation and discussion in section 5; we compare our

---

1 This plugin is available in Google Open: [http://code.google.com/p/smartreplayer/](http://code.google.com/p/smartreplayer/)
work with related approaches in section 6 and end this paper in section 7 with conclusion and future work.

2. An Illustrative Example

In this section, we take the task of checking out a CVS project by Eclipse v3.3 as an example. This is not only a repetitive task to a single programmer who use CVS for development, but a repetitive task common to a team of programmers who cooperate to do some development work. We use this simple example to show how boring and time consuming these kinds of tasks can be and to reveal the significance of automating them in software development.

The following list shows the user actions to check out the JHotDraw project from its CVS repository.

- Select Menu Items “File->New->Other...”. (3 user actions)
- In the popup wizard, select item: “Project from CVS”, then click “Next” button. (2 user actions)
- In the following “Enter Repository Location Information” wizard page, input jhotdraw.cvs.sourceforge.net to the “Host” textfield, input /cvsroot/jhotdraw to the “Repository Path” textfield, and use “anonymous” as the “User Name”. Leave “password”, “CVS connection type”, “Port” and other fields to the default options, then click “Next”. (more than 4 user actions)
- In the following “Select Module” wizard page, choose the “Using an existing module” radiobutton which will drive Eclipse to fetch modules from the CVS repository. Wait until the modules are fetched back and presented by a tree structure. There are three existing modules: JHotDraw, jhotdraw6, and jmodeller. Choose the JHotDraw module, then click “Next”. (more than 5 user actions)
- In the “Check Out As” wizard page, select “Check out as a project in the workspace” radiobutton. Input “MyProject” as the project name and select “Checkout subfolders” checkbox, then go on to click “Next”. (4 user actions)
- Finally, in the “Select Tag” wizard page, there are four kinds of tags from which allow users to select. They are: “Head”, “Branches”, “Versions”, and “Dates”. Choose the “Head” tag, then click “Finish”. After that, this task has been done. Programmers can wait for the project to be checked out, and then continue to work on it. (2 user actions)

We can see that programmers have to perform more than 20 actions in order to finish such a simple task. On average, this task will cost them 2 minutes and 48 seconds. Why wasting the time if this task can be automated by replaying. Besides, programmers may do some editing when repeating this task. For example, they could change the local project name from “MyProject” to “GraphProject”. Why not make them directly edit this text without performing other actions. In other cases, if programmers decide to choose the “Versions” Tag instead of the “Head”, they have to do the preceding 18 actions before getting to the “Select Tag” wizard page, if this page can be made present directly to the programmer, a lot of time and efforts will be saved.

3. Catch and Replay User Actions

In this section, we will discuss how SmartReplayer supports automatically executing the IDE-based tasks by using catch/replay technique. It includes two phases: the catching phase records a programmer’s actions when he/she using an IDE to do a task; the replaying phase automatically performs the recorded actions to repeat that task.

3.1. Catch User Actions

An IDE-based task is executed as a sequence of user actions. Each of these actions is the operation step programmers performed by using input devices, such as the mouse and keyboard, to operate the IDE. As long as the IDE’s initial state stays the same, repeating the same user actions will certainly drive it to do the same task. So, if we can catch all of the user actions on the IDE, we can record the entire task. Here, the question is: how to catch the user actions.

IDEs use events to handle theirs communication with programmers. So we can catch user actions by catching events. The information contained in events is relatively low level, for instance, when clicking a button which currently locates in the center of the screen, we can get the following event: “type=Mouse Click, position= (512, 384)”. If the UI layout be changed, this information may not help us to locate the corresponding button. Besides, it is not easy to be edited. In order to make the recorded user actions more robust to UI changes when replayed and easier to be edited, we should exploit more high-level information from events to represent them.

SmartReplayer records the following four parts as a user action.

1. UI object features. The UI object is the one with which programmers interact at a certain operation step. Take the button object as an example, the features are something like, its class type, its caption, its size, etc.

2. The context of the preceding UI object. For instance, a button is located in the package explorer of an IDE which is now in the debug perspective. SmartReplayer will record the name and type of the debug perspective, the package explorer and so on.

3. The high-level operation on the preceding UI object. SmartReplayer categorizes the events getting from mouse and keyboard operations during the Human-IDE interaction process into a higher level, as shown in Table 1. For instance, the mouse click or key stroke events which occurred on the Button, Menu.Item, or ToolBar.Item are mapped to Selection. In this way, we can reduce the information which needs to be recorded,
and make them hold more semantics than just recording things like KeyDown, KeyUp, etc. It can also benefit programmers who edit the action sequence, because it is clearer and more comprehensible.

4. The time property. It includes the time cost to finish a user action and the interval between two successive actions.

All of these four parts are indispensable. The features and context can be used to locate the corresponding UI objects for replaying tasks; the high-level operation can be used to drive the found UI object to work; the time property can be used to adjust replaying speed. The recorded user actions are more robust to UI changes, which will be discussed more in the related work.

The low-level events can be easily collected by listener [4] or hook [5] technique. For instance, in Eclipse, there is a “Display” class which is designated for event dispatching and forwarding. We can register listeners to it, so as to be notified before or after the occurrence of the events. The four parts of a user action can be deduced from its corresponding events. For instance, after getting an event, SmartReplayer will analyze the event-source object and its parent UI by reflection mechanism [6] to obtain the features and context.

The recorded action sequence can be presented by a tree structure, as shown in Figure 1. The triangle icon presents a user action (the Modify action is presented by the pencil-paper icon to denote that it can be edited); the four-circle icon presents the context. From this figure, we can see that this action sequence presents a programmer checking out files from CVS: he/she first selected the “File”, “New”, and “Other” menu items to bring out the “New” wizard dialog, then in this dialog, he/she expanded the “CVS” tree item to choose the “Project from CVS” sub-item. This kind of nested structure can help to directly display the process of a certain task. Programmers can add explanations in the tag area to every action, as shown in Figure 1, thus to make them more understandable.

### Table 1 Event-Action mapping (excerpt)

<table>
<thead>
<tr>
<th>Low-level Event Type</th>
<th>Event Source Object</th>
<th>High-level Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mouse Click; Key Stroke</td>
<td>Button, MenuItem, ToolBarItem</td>
<td>Selection</td>
</tr>
<tr>
<td>Focus</td>
<td>All of the UI objects which can get focused</td>
<td>Focus</td>
</tr>
<tr>
<td>Mouse Up/Down; Key Up/Down</td>
<td>Text Editor, Text Field</td>
<td>Modify</td>
</tr>
<tr>
<td>Expand; Collapse</td>
<td>Tree, TreeTable</td>
<td>Expansion</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The following describes how to locate a UI object in detail:

- The first step is to sum up the properties of UI classes. We only consider the basic UI classes for the complex UIs are all composed by the basic UIs. We extract some properties which can act as features and store them in a profile document. For instance, to the Button class, its type property like “org.eclipse.swt.widgets.Button” can differentiate itself from other UI classes, its caption, size, and location properties can help differentiate each button object in a given context. All of these properties are features to the Button class. Then, we go on to set a weight to each property to denote its capability to identify the corresponding UI object. We also assign a

### 3.2. Replay User Actions

In order to make the IDE work as expected without human intervention, SmartReplayer first transforms a user action back to the event, then dispatches the event to the IDE which will perform the corresponding operation after receiving an event. This process will be repeated until all of the actions in the sequence have been handled. In this way, the IDE is driven to do a task automatically for programmers.

The first step in replaying phase is to locate the event source object on the UI, so as to impose the event back to it. If every UI object has a Universal Unique ID (UUID), we can thus use it to find the wanted from a lot of UI objects. As mentioned above, UUID should be the one which can remain unchanged permanently no matter how many times the application restarts. To get this kind of UUID, the application must be modified. For instance, the GUI testing tool abbot [11] adopts such an approach to locate UI objects of the tested applications. In order to maintain the independence and integrity of the IDE, we would like to use other technique to locate the UI objects. Here, we propose a method which utilized the feature similarity concept [8] [9] in the image and information retrieval research areas to locate a UI object according to its features. Each UI class has its own properties, for instance, the properties of the Button class include caption (or icon), foreground, background, size, location and so on. To a button object of this class, these properties can act as features which if combined can uniquely identify it. For instance, button A’s caption is “Next”, button B’s caption is “Back”, to locate a button with caption “Next” in a context containing only button A and B, we can quickly find button A by comparing the caption property. The replaying phase also uses this kind of feature-matching principle to locate a UI object.
“feature threshold” to each UI class to determine if a UI object is the wanted one.

- An entry point is needed to begin the UI looking up process. The root UI is sufficient for this role, because it is unique and unchanged during the IDE’s lifetime. We start from the root UI and check its children to find the wanted object.
- When checking a UI object, it is first assigned a “feature score” with the initial value 0. Then according to the features in the profile document, its corresponding properties are checked at the decreasing order of the weights. As long as a property’s value matches with the recorded counterpart, its weight is added to the “feature score”. If the “feature score” exceeds the “feature threshold” of the UI class, then this object is thought to be the wanted one. Otherwise, we begin to check the next property. If the “feature score” is still below the “feature threshold” after all the properties have been checked, it shows that this object is not the wanted one. So, we will go on to check the next UI object until all of them have been checked. By comparing the current context with the recorded counterpart, this locating process can be accelerated.

After locating the wanted UI object, the next step is to transform the recorded user action back to the corresponding event and impose it onto this object. SmartReplayer will (1) first create an event object and set the found UI object as its event-source; (2) then look up Table 1 to get its event-type according to the class-type property of the found object and the high-level operation of the recorded user action; (3) finally, dispatch this event. In this way, the IDE is driven to work automatically.

By catch and replay, SmartReplayer can greatly help programmers do repetitive tasks. We gave it to a programmer in our team who used Eclipse and JHotDraw for development. He was asked to use it to record all his actions occurred during the process of checking out JHotDraw project as described in the illustrative example. After finishing this task, he added some explanations to each action and then saved them. Several days later, JHotDraw released a new version which this programmer wanted to try. This time he used SmartReplayer to repeat the task. He first loaded back the action sequence, then replayed it. SmartReplayer automatically executed this task in 34 seconds (including the time cost to fetch the project modules from the CVS repository). If repeating this task manually, it will cost him 3 minutes and 7 seconds with normal speed. Additionally, programmers could use the break-point mode to do replay, for instance, in the experiment, this programmer wanted to choose the “Version” tag instead of “Head” in the “Select Tag” wizard page when repeating the task, he loaded back the action sequence and navigated to the tag-select action as shown in Figure 2, then clicked the “Run here” command in the popup menu which associated with this action, this command drove other actions to be executed and made the “Select Tag” wizard page present directly to him in a very short time. The programmer told us that SmartReplayer increased his programming efficiency.
4. Edit Recorded User Actions

Performing repetitive tasks does not mean redoing the same operations all the time, programmers often have to make some changes during the process. If these recorded user actions can be edited and replayed, a lot of time and efforts can be saved.

4.1. Types of Editing with Samples

There are four types when editing the recorded user actions:

**Delete Actions.** Programmers often encounter some unexpected problems when they perform an IDE-based task, for instance, an exception will happen when they click an improper button. If these error-prone actions be replayed, they may affect the success rate of the replay. Additionally, programmers may do some actions without any contribution to the whole task, for instance, as shown in Figure 2, the “Back” and “Next” actions within the circle are just these kinds of no-contribution actions. In other cases, programmers could open a dialog and close it immediately; they could even input some characters but delete them right away. These actions will reduce the replay efficiency and make the replay process less understandable. So, in order to promote the correctness and efficiency of the replay, programmers need to filter out these unnecessary actions or so called noise actions.

**Add Actions.** When repeating a task, programmers may choose to perform more actions during the process. For instance, in the illustrative example, programmers may choose to select the “Version” Tag instead of “Head” in the “Select Tag” wizard page, then they can go on to click the “Refresh Tags” button to refresh the “Version” and to choose which version they want to check out indeed. These new actions could be added to the recorded action sequence to reflect the task-repeating process.

**Change Action Order.** Programmers may choose to do some actions before other ones when repeating a task. As shown in Figure 1, the “Modify Host” action was done before the “Modify Repository Path” action when recording them, but programmers may edit “Repository Path” first when repeating the task. So, the order of the recorded actions should also be changed.

**Modify Input Parameters.** It means to edit the text in a given text editor. The text editor includes all kinds of textfields, textareas, source code editors and so on. Because the text in a text editor can be used as the parameters passing to the following operations, this paper calls them “Input Parameters”. Programmers often edit these input parameters during their task-repeating process, for instance, in Figure 2, the highlighted action denotes the programmer inputs “MyProject” as the checked-out project name when performing the task, this programmer may change the project name to “GraphProject” when he/she repeats this task manually, so the corresponding user action should be modified to reflect the change.

![Figure 2](image_url) The end portion of the action sequence from Figure 1. The actions in the red circle can be deleted as noise actions; the highlighted text is the “Input parameter” to the next actions.

In *SmartReplayer*, programmers can use the “Delete” command to delete actions, or use the “Edit” command to modify input parameters, as shown in Figure 1. They can also drag and drop the items in the action-tree to change the action order. Besides, they can add actions directly into the file which contains the action sequence.

4.2. Guarantee Correctness

There should be a way to guarantee the correctness of the sequence of user actions after they have been edited. In the following, we will describe how *SmartReplayer* guarantees the correctness of the four editing types.

**4.2.1. Correctness of Deleting.** The noise actions to be filtered out by deleting include: error-prone actions and no-contribution actions. The filtering criteria for identifying these actions are different based on the goal of the tasks. For instance, if the goal is debugging, the actions of inserting some debug codes but deleting them afterwards are important and can not be filtered out. So, before deleting noise actions, a filtering policy should be supplied according to the goal of the task. However, no matter what filtering policy used, *SmartReplayer* supports two mechanisms to guarantee the correctness of deleting: the first needs to understand the IDE’s semantic; the other does not.

**Semantic-dependent Mechanism.** IDE’s semantics are the meanings of its UIs. For instance, clicking a “Cancel” button means canceling the related operations which have done before; clicking an “Apply” button in a wizard means making all of the operations which have done on this wizard take effect right away. When deleting a noise action in the recorded sequence, some of its semantically-related actions should also be deleted. For instance, in Figure 2, the “Back” and “Next” actions within the red circle are noises which do not contribute to the whole task, if only deleting the “Back” action, some errors may occur when replaying this task. Programmers must be reminded to delete the “Next” together with the “Back” action. *SmartReplayer* uses different algorithms to deal with the noise actions which hold different semantics. The following algorithm describes
how to deal with the “Back” and “Next” actions in the preceding example:
1: Search all of the segments which start with “Window” (it is actually the window context of an action) in the action sequence. Push them into a stack. As these window segments are nested as shown in Figure 1, in the algorithm, all the actions we mean in a given window segment must be directly belong to this window, namely the contexts of the actions are exactly this window.
2: If the stack is not empty, then popup a window segment from it, else go to 7.
3: In this segment, find the “Next” action and then add it to a queue.
4: If comes to a “Back” action and there is no other actions before it besides the preceding “Next” action, then let the “Next” action out of the queue and mark it to denote that it could be pair-deleted with this “Back” action.
5: Go on to execute 3 and 4 until coming to the end of this window segment.
6: Return to 2.
7: Supply these marked actions as a reference to programmers when filtering them.

Semantic-independent Mechanism. The semantic-dependent mechanism will get a relatively higher accuracy when used for deleting noises. But its scalability is limited, because it has to design different deleting algorithms to suit different UIs which hold different semantics. In order to cope with this problem, SmartReplayer supplies an alternative mechanism which does not depend on semantics to filter out noises. It draws on the Input-Output Hidden Markov Model (IOHMM) [13] to do the noise-filtering work on an action sequence. The basic idea is: it first extracts a common action sequence from a lot of action sequences which hold the same task goal, then compares each sequence with this extracted one; if an action in the given sequence cannot be mapped to an action in the common sequence, this action could be deleted as a noise action. The advantage of this mechanism is that it has a relatively higher scalability as it is not confined to semantics. But in order to increase the deleting accuracy, a lot of same-goal action sequences are needed.

4.2.2. Correctness of Modification. In order to guarantee correctly replay a given action sequence after the input parameters in it have been modified, we must make these modifications effective globally. For instance, as shown in Figure 2, the highlighted action shows that a programmer input “MyProject” as the checked-out project name in an Eclipse wizard. Eclipse will then create a project context with the name “MyProject” for him/her. If this name is changed to “GraphProject”, all the actions related to “MyProject” in the action sequence must be changed together. SmartReplayer uses an algorithm to guarantee global-modification. When programmers modify an input parameter, all the related actions found by this algorithm will be highlighted to ask if they need to be modified together. After getting confirmation from the programmer, they will be changed automatically. The following describes the global-modification algorithm.

Algorithm Input:
(1) An action sequence: \( A_1, A_2, A_3 \ldots A_n \), where \( A_i \) \((i \leq k \leq n)\) is a user action SmartReplayer captured in the action-catching phase, the four-parts information of \( A_i \) has been explained in Section 3.1.

Algorithm Output:
All the actions related to \( A_i \), Presenting them to ask for confirmation and automatically modify them afterwards.

Algorithm:
1: Assign a correlation threshold: \( t \) to \( A_i \), Its value is based on \( A_i \)’s recorded features and context. For instance, the \( A_i \) appeared in the context of the “New Project” wizard will often have more significant impact on its following actions than the \( A_i \) appeared in the source code editor, so the former’s \( t \) will be set bigger than the latter’s.
2: Let \( a + \beta + \lambda = 1 \). They are all regulators which with the value in (0, 1).
3: The correlation between \( A_i \) and its following action \( A_j \) is decided by the 3 parts below:

3.1: Sequence correlation: \( s \). \( s \) is calculated as follows:
\[
s = \frac{(n-j+1)\times t \times \alpha}{(n-i+1)}\]
\( s \) will get bigger as long as \( A_j \) get more closer to \( A_i \). Because a modify-action will often have more significant impact on its direct successor than on its indirect successor and the impact will often get reduced if \( A_j \) go further away from \( A_i \).

3.2: Context correlation: \( c \). \( c \) is calculated as follows:
If \( A_j \) and \( A_i \) have the same context, namely, the UI context such as wizard dialog is the same when these two actions happened, then \( c = t \times \beta \); else \( c = t \times \beta / 2 \).

3.3: Operation-Context correlation: \( o \). \( o \) is calculated as follows:
\( o \) represents the correlation between \( A_i \)’s operation and \( A_j \)’s context. Refer to the previous example, suppose \( A_i \) input “MyProject” in the textfield as the project name, if “MyProject” appeared in the context of \( A_j \), then \( o = t \times \lambda \); else \( o = 0 \).

4: If \( s + c + o > t / 2 \), then \( A_j \) is thought to be related with \( A_i \).
5: Mark \( A_j \), let \( j = j + 1 \), then return to 3, until \( j = n \).
6: All of the marked \( A_j \) \((i \leq k \leq n)\) will be checked by the programmer and these \( A_j \) will be automatically modified after they have been confirmed.

4.2.3. Correctness of Adding actions and Changing Action Order. When adding an action, programmers need
to supply the four parts information which makes up this user action, but the information such as the UI object features and context could not be validated until replaying, because in order to promote the recording efficiency, SmartReplayer only records the information of the UI objects which have been operated by programmers. Changing the action order may impact the logic of the action sequence which also cannot be validated until replaying. If SmartReplayer records the entire UI objects in a given context and the relationships of these objects, it can use them to validate any newly added actions and the logic of the order-changed action sequences before replaying. But recording more information requires more processing time which will decrease performance. SmartReplayer uses sequence-comparing to cope with this problem. If adding an action to a given action sequence and there is a similar action recorded in another action sequence which holds the same task goal, SmartReplayer can validate this newly added action by comparing the two sequences. In the same way, the logic of the order-changed action sequence can be validated before replaying. The limitation of this sequence-comparing method is that it needs a lot of same-goal action sequences.

5. Evaluation and Discussion

The current version of SmartReplayer is a research prototype. It handles catch and replay of the tasks which require programmers to operate directly on the UI objects of Eclipse. These tasks can be categorized into three types: (1) Operating wizards and dialogs. Programmers use mouse and keyboard to open a wizard or a dialog, and then work on it. (2) Operating editors. Programmers edit the contents of the editors when performing this kind of tasks. These editors include all kinds of source code editors, text fields, text areas, editable tables, etc. (3) Operating views. A view in Eclipse is a logic unit to organize several UI objects to work together for a specific purpose. For instance, in Eclipse v3.3, there are “Package Explorer View”, “Outline View”, “Hierarchy View”, etc. Programmers can operate the UI objects in these views when performing a task. SmartReplayer supports the three types of tasks and their combinations. However, if the user operations are not directly performed on the UI objects, SmartReplayer cannot catch them based on the mechanism described in section 3.1. For instance, when programmers use the visual modeling function in Eclipse to draw a UML graph, the shape objects which are directly operated by programmers via the mouse do not belong to UIs. So if SmartReplayer only records the actions like mouse-drag and mouse-click, replaying these actions cannot directly modify the corresponding UML model, because the operating objects are not the children of the container UI, but painted directly on the container UI. Future SmartReplayer will create an adapter to interpret the model changes to the user actions so that the visual modeling process can be caught and replayed.

Being a prototype, SmartReplayer has not yet been evaluated extensively, especially the performance. However, some limited performance tests have been conducted. When using SmartReplayer to record a task, the execution time can be divided into two parts: one is spent on extracting the user actions from the low-level events; the other is the normal time programmers spend in performing each operation. The overhead of SmartReplayer was obtained by computing the ratio of the above two parts. We tested 100 tasks in Eclipse v3.3 which include creating EJB projects, building Swing and SWT applications, etc. On average, each of these tasks contained 128 user actions. The average increase of the execution time for these 100 tasks was 0.64% during the action-catching phase and the average size of the recorded action sequences was 15.2KB per 100 user actions. This suggested that the current SmartReplayer design can catch and store user actions efficiently.

We tested the correctness of editing these action sequences, although the editing functions were still under development. On average, we deleted 12 actions and modified 9 input parameters in each action sequence. After manually checking the validated actions presenting by SmartReplayer, we observed that the accuracy of deleting is 94% and the accuracy of parameter-modification is 86%. Adding actions and changing the action order had also been tested. By comparing 20 action sequences which hold the same task goal, 54% of the newly added actions which SmartReplayer suggested to be ok could be replayed correctly and the accuracy of changing action order was 44%. But the accuracy increased to 63% and 58% respectively after comparing 40 same-goal action sequences. This suggested that (1) adding actions and changing action order were mainly relies on programmers’ skills and experiences; (2) validating these actions need a lot of same-goal action sequences.

6. Related Work

The catch and replay technique has been used in several other research areas. But there are some differences between SmartReplayer and them.

The first similar work is the macro. It is used to record mouse and keyboard functions for playback at a later time. The Macro [15] in MS Word is one of its implementation tools. The first difference between Macro and SmartReplayer is that: the information recorded by Macro is relatively low-level and not convenience for editing. For instance, Macro will record something like “Application.Move Left = 96, Top = -3”. To understand and edit this information, programmers must have some knowledge on languages like VBA [16]. In SmartReplayer, the recorded information is like: “Select menu item: File”, “Click: Next”, etc. They are rich in semantics content, and easier to understand and more convenient to edit. The second difference is that: Macro only records and plays back the results of the software artifacts after they have been changed, whereas SmartReplayer also records the change.
process of these artifacts. The change process is actually the user actions to bring about the changes of the software artifacts. Recording these actions can make programmers get more control over the task-repeating process because the process can be more easily edited with these actions.

The second similar work is “Auto GUI Test”. jacareto [10] and abbot [11] are these kinds of tools. Test cases are generated by recording the mouse movements and keystrokes on the UI objects. In a study by Menon and Soffà, 74% of these kinds of test cases became obsolete after a major change to the user interface [14]. One of the main reasons is that the corresponding UI objects cannot be found when reusing these test cases. SmartReplayer uses feature-matching to locate UI objects, thus less sensitive to UI changes. Besides, by using feature-matching, SmartReplayer keeps the integrity of the IDE’s source codes, whereas jacareto and abbot need to insert some codes to the testing programs to aid the UI-locating process.

Catch and replay is also used for technical support. Sheepdog [12] is one of the research work. This tool learns from multiple experts, each performing the same procedure on the same window system. It records execution traces, and uses them to build a model of the procedure that can be executed on a user’s system. Sheepdog relies on experts to demonstrate the action traces; whereas SmartReplayer allows any programmer to create action sequences to do the tasks specific for him/her. Additionally, when recording the action traces, Sheepdog needs to save the snapshots of the “world state” which will incur high overhead in the runtime and will bring unnecessary noises, whereas SmartReplayer records only the information which related to the user actions and the information is enough to automatically execute the repetitive tasks, thus the runtime cost of SmartReplayer is relatively lower.

7. Conclusion and Future Work

We present an approach based on editable visual replay to automatically execute the IDE-based repetitive tasks. The user actions during the task-execution process are caught, and then replayed to reduce the burden of programmers when repeating the task. The user actions can be edited by programmers to meet their personalized needs. We also present SmartReplayer, an Eclipse plugin to implement our approach.

Our future work on SmartReplayer is to improve it and use it in large-scale cases. We also expect to use SmartReplayer to do more things. One future work under consideration is to extract the best action trace which cost least time and efforts to perform a task by mining the recorded user actions. Another potential direction for future work is to use SmartReplayer to do distributed collaboration. Programmers in different work places are expected to collaborate by catching one programmer’s actions and use these actions to help other programmers.

References

[9] Adam Baumberg. Reliable feature matching across widely separated views. CVPR'00 -Volume 1 p. 1774