# Extending Java with Checkpointing

Thomas Huining Feng
Ptolemy Group, EECS, UC Berkeley
tfeng@eecs.berkeley.edu

November 27, 2005

**Abstract**

In this project, an incremental checkpointing mechanism is developed for Java. By making the abstraction that program state is the state of variables, this checkpointing mechanism gives the assurance that the program can be rolled back to its previous state when error occurs. A language construct that ensures this data integrity is invented with a syntax similar to Java's try-catch-finally construct. Transformation from a program written with this new construct to a native Java program is studied. An tool is implemented to automate this transformation.

## 1 The Problem of Maintaining Program State

Exceptions as a language feature are supported by a number of contemporary programming languages. The use of exceptions and the corresponding try-catch-finally construct simplifies error handling in programs.

However, correct use of the try-catch-finally construct does not guarantee correct handling of exceptions. Although the program execution recovers from the exception, the program state may be inconsistent from that point. Figure 1 shows a program fragment in Java. It may invalidate its execution state after the occurrence of an IO exception. This code tries to read from an input stream, and then append the received data to the non-empty buffer. However, if an `IOException` happens in the loop, the execution of the loop will not be complete. Though the exception is caught and handled, the state of the program, which includes `buffer` and `bufIndex` in this case, is no longer consistent. What the programmer really wants is to read the data into the buffer, but if the read is not complete, neither the buffer nor its index should be changed, as if no read operation were issued.

A simple-minded attempt to correct the problem results in another program in Figure 2. This time, a full backup of the program state is created before the try block is entered. If an `IOException` occurs, the program state is restored with the backup. However, this modification only partly solves the problem. The `(*)` statement is problematic. If the reference of `buffer` was copied to another variable before,

```
int[] buffer = ...  // A buffer
int bufIndex = 0;   // The current index of the buffer
...
bufIndex = ...      // Current location of the buffer (not empty)
...
try {
    InputStreamReader reader = ... // An input stream reader
    int i; // Variable to hold input value
    do {
        i = reader.read();
        if (i != -1)
            buffer[bufIndex++] = i; // Add the input to the buffer
    } while (i != -1); // Until EOF
} catch (IOException e) {
    ...
}
```

Figure 1: A simple program that may invalidate its state in case of exception

```
int[] buffer = ...  // A buffer
int bufIndex = 0;   // The current index of the buffer
...
bufIndex = ...      // Current location of the buffer (not empty)
...
int[] backupBuffer = (int[])buffer.clone();
int backupBufIndex = bufIndex;
try {
    InputStreamReader reader = ... // An input stream reader
    int i; // Variable to hold input value
    do {
        i = reader.read();
        if (i != -1)
            buffer[bufIndex++] = i; // Add the input to the buffer
    } while (i != -1); // Until EOF
} catch (IOException) {
    ...
    buffer = backupBuffer; // (*)
    bufIndex = backupBufIndex;
}
```

Figure 2: The modified version of the simple program that may still violate its state

```
STATEMENT ::= ... | BLOCK | SAFE
BLOCK ::= { STATEMENT* }
SAFE ::= safe BLOCK
```

Figure 3: Partial language definition of the safe block

that variable still points to the old buffer after `(*)`, so the program state is still inconsistent. A correct solution would require `(*)` to be replaced with: `System.arraycopy(backupBuffer, 0, buffer, 0, backupBuffer.length);`.

However, even if the final solution to this simple example is found without much difficulty, we may still question the practicality of this ad-hoc approach in general. First of all, this requires the programmer to identify the program state that should be kept consistent, and then write the correct code to maintain this consistency. If this is not done correctly, the erroneous result may not be observable immediately, leading to a mysterious crash at a later time. Secondly, the correct code is too inefficient. It requires a copy of the whole array every time the try block executes successfully, and two copies in case of an `IOException`. If the array is large and the update in the try block affects only a relatively small area, much of this copying is wasted.

# 2 A Language Approach

The fix in the last example is very ad-hoc. If the program is large, no correctness assurance can be given. In this section, we will invent a new Java construct that automates error handling and provides the assurance.

## 2.1 A syntax and an informal semantics

If we review the try-catch-finally construct in Java, it is not hard to see that it only regards program control, while ignoring the data. Because programs are made up of data and control, a new language construct that takes data into account will compensate for this.

Figure 3 shows part of the language definition with a construct called "safe block" added. A statement in this language can be any legal Java statement (including block), or a safe block. A block is a compound statement that contains 0 or more statement between curly brackets. A safe block contains a block as its body, preceded by the "`safe`" keyword.

To explain its semantics informally, we first take a look at the control flow. The program execution enters the safe block naturally when it is reached. However, there are two paths on which a safe block is exited. If all the statements in the block complete successfully (i.e., all the exceptions, if any, are caught), the block is exited normally. On the contrary, if an exception occurs in one of the statements

and it is not caught by any catch clause, the execution of the safe block terminates, and the control is returned to the surrounding context. In this sense, the safe block does not affect the program control, as if it were just a Java block. However, a safe block differs from a block in that it guarantees that the program state is restored if the control leaves with an exception. By this means, invariants about program state are maintained.

### 2.2 A formal semantics

A more formal semantics requires a definition on program state at first. In this work, we define program state as such:

> *Program state at a time in execution is the transitive closure of all the* `private` *non-*`static` *variables visible to the program at that time.*

We do not consider local variables in program state, because they are usually transient, and we can always turn them into `private` variables (possibly with renaming). We do not consider `public` or `protected` variables to be program state, either. This is because it is generally not a good practice to keep state that needs to be consistent with `public` or `protected` variables. To allow access from other classes while keeping them `private`, we can add *getter* and *setter* methods for them.[1]

In operational semantics, we use $\sigma \in \Sigma$ to denote program state. We do not consider creation of variables, because we can regard an assignment to a fresh variable to be an update on the variable with a default value. Function $except(\sigma) : \Sigma \rightarrow \{true, false\}$ takes a program state as parameter, and returns $true$ if there are uncaught exceptions in the state, or $false$ otherwise. Remind that the operational semantics of command $c$ is denoted with $< c, \sigma > \Downarrow \sigma'$, where $\sigma$ is the initial state, and $\sigma'$ is the new state after the execution of $c$. The operational semantics of the block construct is:

$$\frac{< c_1, \sigma > \Downarrow \sigma'' \quad except(\sigma'') = false \quad < c_2, \sigma'' > \Downarrow \sigma'}{< c_1; c_2, \sigma > \Downarrow \sigma'} \tag{1}$$

$$\frac{< c_1, \sigma > \Downarrow \sigma' \quad except(\sigma') = true}{< c_1; c_2, \sigma > \Downarrow \sigma'} \tag{2}$$

$$\frac{< c_1; c_2; \ldots; c_n, \sigma > \Downarrow \sigma'}{< \{c_1; c_2; \ldots; c_n\}, \sigma > \Downarrow \sigma'} \tag{3}$$

The operational semantics of the safe block can thus be defined as:

$$\frac{< \{c_1; c_2; \ldots; c_n\}, \sigma > \Downarrow \sigma' \quad except(\sigma') = false}{< safe\ \{c_1; c_2; \ldots; c_n\}, \sigma > \Downarrow \sigma'} \tag{4}$$

$$\frac{< \{c_1; c_2; \ldots; c_n\}, \sigma > \Downarrow \sigma' \quad except(\sigma') = true}{< safe\ \{c_1; c_2; \ldots; c_n\}, \sigma > \Downarrow \sigma} \tag{5}$$

Rule 4 applies when the safe block is executed without any uncaught exception. In that case, the new state of the program is the same as the new state $\sigma'$ returned by the block. If uncaught exceptions are found in the new state $\sigma'$, rule 5 enforces a *rollback* to the initial state $\sigma$, as if the commands in the block were not executed.

## 3 Implementation Mechanisms

The formal semantics defined in the last section does not specify how the safe block is implemented. Rule 5 is a strange one because it requires the state after the execution of the safe block to remain the same as the initial state $\sigma$, even if some commands in the block are executed successfully. I.e., the new state of the safe block cannot be derived from the pre-conditions over the line in rule 5. The occurrence of exceptions in the block is unpredictable. For a sequential program to be executed according to this semantics, there must be some mechanism to make sure that it is possible to rollback to the initial state once a safe block is entered.

[1] Later in this paper, we will use an automatic analysis tool to analyze change of program state. If we include `public` or `protected` fields in program state, the analysis becomes much more complex, since inter-class access must also be considered. This is another reason for restricting our scope to `private` fields.

Checkpointing technique can be used as an implementation mechanism. When the safe block is entered, a checkpoint is created, which refers to the program state at that time. Later, the checkpoint created earlier can be used to restore the program state in case of exception. There are different checkpointing strategies. We will first discuss snapshot checkpointing, and then elaborate on incremental checkpointing.

## 3.1 Snapshot checkpointing

An obvious approach is to take a snapshot of the program state at the time when a safe block is entered. The simulator or the executing program keeps the snapshot at a secret place so that the commands in the block will not modify it. If an uncaught exception occurs, the snapshot is used to restore the program state; otherwise, the snapshot is simply discarded. It is convenient to store the snapshots in the stack, because this allows hierarchical safe blocks in subsequent method invocations.

A problem of this approach is identifying the range of a snapshot. A snapshot should include all the variables that may be modified in the block. It is safe to create snapshots for all the declared variables, because the set of modified variables always form a subset. However, this results in very inefficient execution. It is also unnecessary in most cases, because only very few of them are actually modified. It is possible to reduce the snapshot size if we can predict which variables will be modified, or if we can find a smaller superset of the modified variables. This can be partly achieved by using a static program analyzer. Unfortunately, because static analysis is conservative, the snapshot size is still very big.[2]

## 3.2 Incremental checkpointing

Compared to snapshot checkpointing, incremental checkpointing has a different flavor in that it does not require a snapshot at the time when checkpoint is created. At the checkpoint creation, virtually nothing needs to be done, except that a flag is set with the current execution *timestamp*. When the variables are modified in the safe block, the modifying operation is captured in some way, and the old values are backed up right before the modification. I.e., the backup of the program state is carried out incrementally as the commands are being executed.

Let us now consider the assignment statement. In operational semantics, assignment can be defined with the following rule:

$$\frac{< e, \sigma > \Downarrow n}{< x := e, \sigma > \Downarrow \sigma[x := n]} \tag{6}$$

This rule means, if expression $e$ is evaluated to number $n$ in state $\sigma$, then the new state after executing command $x := e$ in state $\sigma$ is the same as $\sigma$, except that the value of $x$ becomes $n$.

We now introduce a new symbol $\varphi$ to denote the state of our checkpoint object. A new checkpoint object is created every time we create an incremental checkpoint. $\varphi[x := n]$ means to record $x$'s old value $n$ in the checkpoint object before $x$ is modified. We redefine the assignment rule with $\varphi$ added to the state:

$$\frac{< e, (\sigma, \varphi) > \Downarrow n \qquad \sigma(x) = n_0 \qquad \varphi(x) = undefined}{< x := e, (\sigma, \varphi) > \Downarrow (\sigma[x := n], \varphi[x := n_0])} \tag{7}$$

$$\frac{< e, (\sigma, \varphi) > \Downarrow n \qquad \varphi(x) = n_0}{< x := e, (\sigma, \varphi) > \Downarrow (\sigma[x := n], \varphi)} \tag{8}$$

In these rules, we extend the program state from $\sigma$ to a tuple $(\sigma, \varphi)$. In rule 7, we define that if the expression $e$ evaluates to $n$ in state $(\sigma, \varphi)$, and the old value of $x$ is $n_0$, and the old value of $x$ has not been recorded in $\varphi$, then the new state after the assignment is the same as $(\sigma, \varphi)$, except that $\sigma(x)$ becomes $n$, and $\varphi(x)$ becomes $n_0$. According to rule 8, if the old value of $x$ has been stored in $\varphi$, only $\sigma$ changes after the assignment. Note that in such contemporary languages as Java, evaluation of an expression is not side-effect free. However, we can safely ignore this aspect here.

The rule for the safe block is then redefined with the checkpoint state added ($\varphi_0$ here means a fresh empty checkpoint object):

$$\frac{< \{c_1; c_2; \ldots; c_n\}, (\sigma, \varphi_0) > \Downarrow (\sigma', \varphi') \qquad except(\sigma') = false}{< safe\ \{c_1; c_2; \ldots; c_n\}, (\sigma, \varphi) > \Downarrow (\sigma', \varphi + (\varphi' - \varphi))} \tag{9}$$

[2] E.g., the analyzer usually cannot tell which elements in an array are modified, so a full backup of the whole array is required to make this checkpointing sound.

Before transformation:

```
private int i;
...
    i = 1;
...
```

After transformation:

```
private int i;
protected Checkpoint $CHECKPOINT = new Checkpoint(this);
private FieldRecord $RECORD$i = new FieldRecord(0);
...
    $ASSIGN$i(1);
...
private final int $ASSIGN$i(int newValue) {
    if ($CHECKPOINT != null && $CHECKPOINT.getTimestamp() > 0)
        $RECORD$i.add(null, i, $CHECKPOINT.getTimestamp());
    return i = newValue;
}
```

Figure 4: Program transformation for a simple assignment

$$\frac{< \{c_1; c_2; \ldots; c_n\}, (\sigma, \varphi_0) > \Downarrow (\sigma', \varphi') \qquad except(\sigma') = true}{< safe\ \{c_1; c_2; \ldots; c_n\}, (\sigma, \varphi) > \Downarrow (\sigma, \varphi)} \tag{10}$$

Rule 9 uses expression $\varphi + (\varphi' - \varphi)$ to compute the new checkpoint state after the block is exited successfully. $\varphi'$ is the checkpoint state that the block returns. $\varphi' - \varphi$ is the part of checkpoint state that is not included in $\varphi$. This part of checkpoint state records the new updates to variables within the block.

Rule 9 and rule 10 look similar to rule 4 and rule 5. However, the extended rules make clear how safe blocks can be implemented.[3] In rule 10, with the $\varphi'$ that the block returns, it is possible to roll back the new program state $\sigma'$ to $\sigma$. The old values recorded in $\varphi'$ are restored to the variables, so that no difference in program state can be seen in the future.

# 4 Program Analysis and Transformation

To implement safe block as an extension to the Java language, source-to-source program transformation is taken as an approach. Programs using safe blocks are analyzed and transformed into native Java code. The transformed code will then be compiled with the Java compiler and be executed in the Java Virtual Machine.

## 4.1 Analysis of program variables

Before transformation can be carried out, the program must be analyzed so that the variables in the program state are discovered and typed. Because we only consider `private` non-`static` variables here, we can focus on one Java source file at a time. In Java, `private` variables can only be accessed by the class that defines them, the classes enclosed in that class, and the classes enclosing that class. The analyzer first resolves types for all the object references, and then decides whether an assignment in the file is a modification on one of the `private` variables. This step is very similar to the type check and the access control [1] processes in the Java compiler.

## 4.2 Transformation of assignments

An assignment to a `private` variable is transformed into an equivalent method call. The method, before actually assigning the variable, backs up its old value in the current checkpointing object (if it has not been backed up yet). This corresponds to rule 7 and rule 8.

[3] Rules 9 and 10 are also applicable to an implementation with snapshots.

Before transformation:

```
private int[] a;
...
    a = new int[10];
    a[i] = 1;
...
```

After transformation:

```
private int[] a;
protected Checkpoint $CHECKPOINT = new Checkpoint(this);
private FieldRecord $RECORD$a = new FieldRecord(1);
...
    $ASSIGN$a(new int[10]);
    $ASSIGN$a(i, 1);
...
private final int[] $ASSIGN$a(int[] newValue) {
    if ($CHECKPOINT != null && $CHECKPOINT.getTimestamp() > 0)
        $RECORD$a.add(null, a, $CHECKPOINT.getTimestamp());
    return a = newValue;
}
private final int $ASSIGN$a(int index0, int[] newValue) {
    if ($CHECKPOINT != null && $CHECKPOINT.getTimestamp() > 0)
        $RECORD$a.add(new int[]{index0}, a[index0], $CHECKPOINT.getTimestamp());
    return a[index0] = newValue;
}
```

Figure 5: Program transformation for array assignments

Figure 4 shows the transformation for an assignment to integer variable `i`. Among the new fields and methods added to the code, new fields `$CHECKPOINT` and `$RECORD$i`, and new method `$ASSIGN$i` are the most important ones.[4] The assignment is replaced by method call `$ASSIGN$i(1)`. If no checkpoint has been created (in which case `$CHECKPOINT.getTimestamp()` returns 0 or negative), the method `$ASSIGN$i` behaves exactly the same as an assignment; otherwise, it backs up the old value before the assignment. (If a backup already exits for the current timestamp, `$RECORD$i.add` returns immediately.)

`$RECORD$i` is the record for variable `i`. It is formed as a stack, so the creation of a new checkpoint results in a new level in the stack. Different levels are ordered by the timestamps about when they are created. An old value in the stack can be uniquely located with the timestamp.

Finally, a checkpoint object called `$CHECKPOINT` is defined for each object in the program. Different checkpoint objects share the same timestamp system, so that their timestamps are comparable. This is necessary because the rollback of one object usually requires the rollback of other objects that it refers to.

The transformation for array assignment, shown in Figure 5, is similar. In this case, parameter `1` is given to the constructor of the record, meaning that the variable to be taken care of is a one-dimensional array. Because `a` is assigned in two ways, two corresponding methods are created. Method `$ASSIGN$a(int[])` backs up the array variable when a new array is assigned to it. Unlike the snapshot mechanism, this backup only requires a copy of the array reference but not the whole array. Method `$ASSIGN$a(int, int)` regards its first argument as the index of the element to be modified, and it creates a backup for that element only. Together with the element's old value, the index is also recorded, so that later we can restore the old value to that index.

Figure 6 shows a more complicated example of this transformation. In this example, an object is declared `private`, so it is regarded as part of the program state. The assignment to modify this object is transformed into an invocation of `$ASSIGN$o`. Unlike other assignment methods, this method also ensures that the checkpoint object used by `o` is the same as the one used by the current object. The checkpoint object then monitors two Java objects concurrently. A rollback with that checkpoint object will be effective on both the two Java objects. This example also contains a statement `i++`. It modifies `i` in a special way. In the transformed code, this becomes a call to `$ASSIGN$SPECIAL$i`, which handles all such special expressions with side-effect.

We transform all the assignments in this way. Because we do not consider side effects other than

[4] As a convention, the new identifiers introduced by the transformer always start with a dollar sign to avoid conflict.

Before transformation:

```
private MyObject o;
...
    o = new MyObject();
    o.changeState();
...
class MyObject {
    private int i;
    ...
    public void changeState() {
        i++;
    }
}
```

After transformation:

```
private MyObject o;
protected Checkpoint $CHECKPOINT = new Checkpoint(this);
private FieldRecord $RECORD$o = new FieldRecord(0);
...
    $ASSIGN$o(new MyObject());
    o.changeState();
...
private final MyObject $ASSIGN$o(MyObject newValue) {
    if ($CHECKPOINT != null && $CHECKPOINT.getTimestamp() > 0)
        $RECORD$o.add(null, o, $CHECKPOINT.getTimestamp());
    if (newValue != null && $CHECKPOINT != newValue.$GET$CHECKPOINT())
        newValue.$SET$CHECKPOINT($CHECKPOINT);
    return o = newValue;
}
...
class MyObject {
    private int i;
    protected Checkpoint $CHECKPOINT = new Checkpoint(this);
    private FieldRecord $RECORD$i = new FieldRecord(0);
    ...
    public void changeState() {
        $ASSIGN$SPECIAL$i(11, 0); // Argument 0 is not used.
    }
    ...
    private final int $ASSIGN$SPECIAL$i(int operator, long newValue) {
        if ($CHECKPOINT != null && $CHECKPOINT.getTimestamp() > 0)
            $RECORD$i.add(null, i, $CHECKPOINT.getTimestamp());
        switch (operator) {
            case 0:  return i += newValue;
            case 1:  return i -= newValue;
            ...
            case 11: return i++;
            ...
        }
    }
    public final Checkpoint $GET$CHECKPOINT() {
        return $CHECKPOINT;
    }
    public final Object $SET$CHECKPOINT(Checkpoint checkpoint) {
        ... // Set the checkpoint object; push a new level to all the records.
        return this;
    }
    ...
}
```

Figure 6: Program transformation for object assignment

Before transformation:

```
try {
    safe {
        ...   // (1)
    }
} catch (...) {
    ...       // (2)
} finally {
    ...       // (3)
}
```

After transformation:

```
try {
    long $CHECKPOINT_HANDLE = $CHECKPOINT.createCheckpoint();
    boolean $SUCCESS = false;
    try {
        ...   // (1)
        $SUCCESS = true;
    } finally {
        if ($SUCCESS)
            $CHECKPOINT.commit($CHECKPOINT_HANDLE);
        else
            $CHECKPOINT.rollback($CHECKPOINT_HANDLE);
    }
} catch (...) {
    ...       // (2)
} finally {
    ...       // (3)
}
```

Figure 7: Program transformation for a safe block

variable update (e.g., disk IO, data transfer to remote devices, screen display, etc.), the only way to change program state is by assignments. Therefore, we successfully capture all the state change.

## 4.3 Transformation of safe blocks

With the assignments correctly transformed, all that needs to be done for a safe block is to create a checkpoint at its starting point, and to roll back to the checkpoint if any uncaught exception occurs, or to commit the state change otherwise.

Figure 7 shows an example of this transformation for a safe block.

As a syntactic sugar, the programmer may write a safe block directly in place of a try block, possibly followed by catch clauses and a finally block as follows. If exception occurs, the state is restored before any catch clause or the finally block is executed.

```
safe {
    ...
} catch (...) {
    ...
} finally {
    ...
}
```

## 4.4 Time and space efficiency

This incremental checkpointing mechanism is designed for high performance. It may even be used in real-time systems. The creation of a checkpoint object at the entry of a safe block takes constant time. The extra code to back up old value for each assignment also takes constant time. At the exit point of a safe block, either commit or rollback is called. The commit operation is linear in the number of variables that are actually modified in the block. For every such variable, a copy of its old value is stored. The commit operation discards the references to those old values. The Java garbage collector

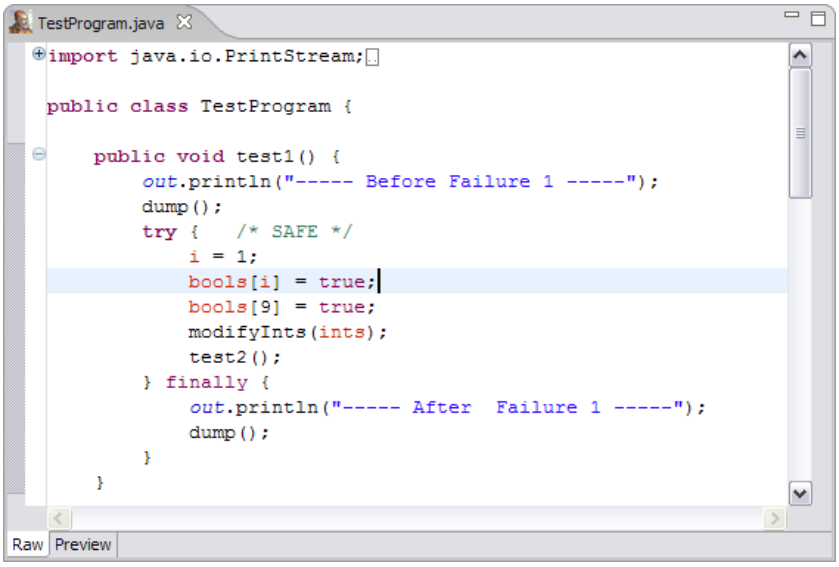


Figure 8: An Eclipse plugin that automates the transformation

will reclaim the memory later. The rollback operation is also linear in the number of modified variables. It first restores the old values to the variables, and then discards the references.

Because assignments are transformed into method calls, constant overhead is added. For a program that uses assignments heavily, this overhead cannot be ignored. To compensate for this, all such methods are declared `final`. An optimizing compiler may inline them at the call sites to eliminate method calls. It is not easy to do this inlining in the transformation process, mostly because Java allows assignments to be sub-expressions in larger expressions. In addition, because method inlining increases code size, it is not always desirable.

Both the snapshot approach and this incremental approach are linear as a whole. However, this approach is still more efficient, because snapshot checkpointing is linear in the number of variables that *may* be modified. The analyzer that snapshot checkpointing depends on always returns a conservative superset of variables that are actually modified at run-time, and this superset is usually much larger.

The memory space that this mechanism requires is also linear in the number of modified variables.

# 5 The Transformation Tool

A tool has been implemented as an Eclipse plugin that automates this transformation process (Figure 8). The Java editor is extended with two views. In the "Raw" view, the programmer inputs his/her program in the extended Java language. Safe blocks can be entered in a slightly different syntax. Instead of using the "`safe`" keyword, a safe block is written as a try-catch-finally construct with the comment "`/* SAFE */`" attached to it. This is purely an implementation issue. Because it is not possible to modify the Eclipse JDT parser, the Java syntax has to be followed, so the "`safe`" keyword cannot be added. When the program is entered, the programmer can preview the transformation result in the read-only "Preview" view.

The plugin also provides a special syntax highlighting in addition to Eclipse's Java source highlighting. All the `private` non-`static` variables are colored orange to remind the programmer that they are regarded as part of the program state. This checkpointing mechanism thus gives the programmer the guarantee that all the orange-colored variables will be safe after an uncaught exception is raised in a safe block.

# 6 Related Work

It is important to maintain program invariance so that the program never runs into an inconsistent state. Exceptions can result in inconsistent program state if the programmers fail to handle them correctly. This effect on critical resources has been explored by the CCured team at EECS, UC Berkeley [2]. They identify program state as the set of resources that are allocated but not released yet. Hence, the operations of resource allocation and deallocation are concerned with. The programmer needs to provide a description on the types of resources and the related operations. A source analyzer takes the program and this description as input, and analyzes all the execution paths, including the paths with exceptions. If there exists any path in which certain resources are allocated but never released, a compile-time warning is produced. This analysis is complete but unsound because false alarms cannot be avoided. Effort has been spent to reduce the number of false alarms.

Compared to this work, the checkpoint mechanism implemented in this project defines program state differently. The program state is the set of `private` non-`static` variables here. The only operation that can modify program state is assignment. Programmers need not provide a description in this case. Source-to-source transformation is employed to capture state change. Instead of informing the programmer potential invalidation of program state, the tool developed here automatically recovers program state when uncaught exceptions occur. It is intended that programmers exploit this safe assurance and ignore some execution paths with exceptions. This releases programmers from the technical detail of restoring programming state every time, and hence simplifies their work.

Because it has many applications, checkpointing is also studied by other research groups. The checkpointing research at Tennessee is developing run-time checkpointing at the Java byte-code level [3]. This checkpointing approach is similar to snapshot. When requested, the complete state of the program execution is recorded in an architecture-independent format. This record can be used to resume the execution, possibly on a different architecture that supports the same Java run-time environment. Because of the nature of snapshot checkpointing, it is not suitable to be used as an implementation technique for safe block. Safe block as a language construct requires frequent checkpoint creation and rollback without much performance penalty.

# 7 Future Work

This project and the work done by the CCured team have different focuses. The former one does not consider resources as program state, so uncaught exception may still do harm to program execution. The latter one does not consider variables, but only checks the allocation and deallocation of resources. It will be nice if the two can be combined, and give programmers better assurance. E.g., when a safe block is exited with an exception, not only the variables are restored, the resources allocated in the block are also released with corresponding deallocation methods automatically.

Another extension to this work is to provide more checkpointing-related language features. With these features, programmers can manually specify the variables in program state, or fine-tune the checkpointing process.

# References


[1] Anne Anderson. Java$^{TM}$access control mechanisms. Technical report, Sun Microsystems, Inc., March 2002.

[2] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pages 419–431, October 2004.

[3] James S. Plank. Checkpointing java. http://www.cs.utk.edu/ plank/javackp.html.