CHAPTER 4

CERTAIN IMPROVEMENTS IN SRASG FOR AUTOMATIC
CONVERSION OF LEGACY CODE

4.1 INTRODUCTION

Rules have been framed to evaluate the legacy system which includes directly retrieves the specification of a legacy system from the new system and evaluates the specification for various modules. This method is easy to access and give efficient results in target system compared to direct Reengineering. This method is implemented to improve the quality of migrated systems by reducing the various defects in the migrated system. In this chapter the legacy system evaluation and conversion is made through Automatic Specification Evaluator (ASE).

4.2 MODULES DESCRIPTION

The main aim of the Software Reengineering (SR) is reusability. The legacy system is the one where the system is not supported and compatible with the new, modern environment. So, there is a need to migrate the older system to a new system which is otherwise known as Forward Engineering. When the components migrated to a new platform, there are several challenges such as the assembling of new applications with their requirements is extremely tedious. Consequently, adaptation and integration do not perform effectively when the SR technique is directly implemented (Kim et al 2001). Due to these constraints, it is not possible to implement the direct
Reengineering technique in the older system. All the converted systems need some structured arrangement to implement the new system to overcome the various issues.

Nowadays reengineering techniques are widely needed for services like web technologies, business and enterprise technologies. In organizations, legacy systems are valuable where each and every module plays a vital role. Direct implementation of SR in this legacy system leads to several problems. The entire software products constantly change because of update and regularities measures. Because of these change factors in a system, the new system should not adopt the legacy system requirements and peculiarities. Apart from these, the method and size also constantly change in real time situations. Despite of many methods, there is no perfect and accurate method of bug free migration.

When the legacy system is converted into a new one, the primary problem is interference. There are several reasons for interferences as one of them is changes of updating between legacy and new system. Engineering changes take place in the new system and these modifications affect the new system behavior semantically and also in other system measures (Verhoef 2000). These changes are indicated as interferences. There are a number of factors that have made parallel development an increasingly serious and critically important problem.

- With the ever increasing size of software system comes the fact that developers must work in parallel to meet market time pressures and schedules. This fact has always existed in large-scale software system development.

- Increasing globalization results in the additional factor of parallel geographically distributed software development (Bin Xu 2005). Temporal and geographical separation makes it
extremely difficult to support the critical informal interactions that provide a significant means of problem solving (Jue-Feng Li et al 2006).

- With the increasing size of systems as they evolve over time comes the fact that only an increasingly smaller proportion of a system is changed for any one.

- Finally as features and feature ownership increasingly drive software systems evolution, we find that software systems evolve on the basis of many independent as well as interdependent software developments. This emphasis the heterogeneity rather than the homogeneity of software systems evolution, and the centralization of the knowledge of changes.

Software systems evolve over time as a result of the requirement changes. The resulting systems tend to have a rich and complex structure, which is highly coupled. Due to long term maintenance and evolution, the documentations may not be able to reflect the actual structure of legacy systems. In many cases, effective partitioning or re-partitioning is needed to decompose the legacy systems with a high cohesion and low coupling principle. The evaluation of the current legacy system is essential for further reuse, because the legacy system normally has been used and maintained for a long time. This evaluation reveals the current status of a legacy system and specifies what phase it is in its Lifecycle.

### 4.3 METHOD TO EVALUATE AND CONVERT THE LEGACY SYSTEM

To implement an engineering technique for an organization, there is a need that it requires enormous data and schedules. There are some critical
information and structures which is very difficult to migrate from legacy systems (Jean Henrard et al 2002). There are some similarities and distinct between the new one and old ones. So maintaining the balance between the old and new system is difficult. To overcome these issues, we proposed a method namely Automatic Specification Evaluator (ASE) for enhancing the migrated new system considering the efficiency and accuracy as the migration factors.

Scoping and planning gives the scope of performance study of the existing legacy IS. The production system is used to monitor the evaluation of the system. It also analyses the behavior of the system during the production. Benchmarking is for testing the system for various data load conditions even if the number of data is expected to load in future. This is also used to attain the gain the attention into the system performance characteristics. Data analysis and correlation collect all the data gathered from the previous stage i.e., monitoring and benchmarking activities. The collected data will be utilized for modeling purposes. The modeling is used to validate the production and benchmarking measurements. If any data peculiarity identifier is there, then that is removed and normalized by the modeling activity. The baseline validation and modeling combine and used for the refinement of the performance and prediction.

When sufficient confidence about the model is achieved, this will be used to predict the system performance and future load scenarios that are beyond the reach of the currently available hardware resources. The legacy system performance evaluation is harder for capacity planning when there is an abundant load growth in the existing IS. This ASE method directly retrieves the specification of a legacy system from the new system and evaluates the specification for various modules. This method didn’t give accurate results but it is easy to access and give efficient results in target
system compared to direct reengineering. This method improves the quality of migrated systems by reducing the various defects in the migrated system.

### 4.3.1 Conversion Technique Using ASE

ASE will reduce the interferences and enhance the new system performance behavior. The data and some functions didn’t compatible with each other and so they get overlapped with each other. This overlapping introduces some serious defects in Reengineering such as loss of data, security measures. ASE identifies the various interferences also reduces them smartly and the resultant system is referred as a new target system.

ASE method evaluates the specification the lower low level by introducing some granules in the new system and gives some breakpoints in the new system too. In general, a system has many attributes and peculiarities. To evaluate the migration efficiency, we check these attributes whether they are correct as that of the legacy system. Our ASE method easily obtains these data without complications. Further the categorization as positive and negative enhancement of this method as well as the target system. To get a fine process we also iterate the process. The below Figure 4.1 gives the overview of the ASE process.

![Figure 4.1 Mechanism of ASE](image)

**Figure 4.1 Mechanism of ASE**
When the legacy system is converted to the new system using any reengineering technique then it undergoes our ASE process and finally we will get the desired accurate new system as the target system. ASE takes place prior to the execution of the new system which is an added advantage of the proposed system. This ASE process is operated in all environments and also suited to integrate in any SR technique. It is quite easy to simulate this ASE as a tool by using packages that include some functions as libraries.

### 4.3.2 Design Methodology of ASE

The steps in the ASE process are summarized in the Figure 4.2. During the operation of ASE process, it should be ensured that the legacy system and its target system memory locations because the execution ASE takes place directly to the memory location. It is easy to setup the desired location of execution as of the user criteria. The tools for garbage collection is optional to delete the no longer required memory areas also they help for common errors. It is possible to break the proposed ASE in any modules as it is a linear and conquer approach.

![Figure 4.2 Process of ASE](image-url)
When the new system obtained from any reengineering technique, the first step is to retrieve the attributes of those systems. When the new system is simple, then it is easy to obtain the attributes by means of manual process like debugging. If it is of large scale having enormous data, then the several tools and metrics are available to retrieve the attributes of the new system. Each and every action of the old system procedures and components are referred as attributes. (Examples of attributes are statements, operations etc.) Moreover these attributes are the main components of the legacy system. We have to ensure that all the available components are reengineered to the new system. Any slicing algorithm or any checking tools should be applied for the reengineering process.

By obtaining these attributes, we are able to detect the interferences in the new system. There are few methods available to identify the interferences in the old system but the ASE method with SR is a promising approach. For detecting the interferences in a system, the attributes are analyzed thoroughly. Here, sufficient time must be spent to analyze the attributes. By doing so it is easy to find the changes and redundancy in the new system.

The two different parallel changes that are delineated in our approach to detecting interference:

- Concurrent changes are changes performed through an optimistic version control system which allows multiple valid copies of the same file at the same time (Meir Lehman 1998). The final version is a semi-automated integrated version. The merging process is usually dependent on the absence of syntactic and direct conflicts. Indirect semantic interference may easily go undetected. Changes of this sort are truly
parallel and there is no assumed ordering among them. Concurrent modification poses several additional challenges in detecting interfering changes.

- Sequential parallel changes are also parallel change in a logical sense. They take place independently from each other and are committed by different developers in a relatively short span time. In a pessimistic version management system parallel changes are sequentialized because only one developer may have a file checked out at a time. The time of submission for a change determines the ordering of the changes. By default, every change is assumed to possibly conflict with previous changes, potentially disrupting or modifying the effects of previous modifications, without notifying any of the developers.

When identifying the redundancy, necessary action should be taken to modify such as deletion and conversion of the new process. After the modification, we should derive and generate a model system so as to perform the migration. Once the model is generated, the model system should be allowed to undergo and check design cases with positive and negative cases inbuilt with ASE. The methods like dependency graph (Anthony et al 2006) or any efficient measurement metrics is required to design the positive and negative cases. The new system falls under the positive case, then execute this system as a target system if it satisfied the user requirements.

The ASE process removes the interference from the new system but still there are some redundancy remains in the target system. This situation arises because of overlapping and inter-dependencies of system components (Akers et al 2005). When the proposed ASE method is again applied to the
system, the interference is further reduced as much as possible. In many cases, the ASE gives the redundant free new system. The mechanism of recursive ASE is given in Figure 4.3.

![Figure 4.3 Mechanism of Recursive ASE](image)

Comparing to the direct reengineering methods, implementation of this ASE is quite time consuming. But when we consider some large scale process and systems, the manual debugging and other corrective measures should take enormous time to complete the debugging process successfully. In this connection, the proposed ASE is an automated task and it will complete the process successfully with minimum time when compared to manual debugging.

### 4.4 IMPLEMENTATION OF ASE

ASE is experimented where CPP considered as Legacy and Java as a new system. So, we retrieve a Java program from a CPP program. For the experiment using ASE, the sample programs in CPP that were used in the previous chapter and ASE generates 18% of program as negative cases which also efficiently removed after implementing ASE. There are various issues arising when the source code is converted to Java. Before the implementation of this method, the importance of the older CPP program was considered and new Java functionalities so as to recover the program without loss and corrective measures. Consider the following CPP code where it undergoes an reengineering program with ASE approach.
Table 4.1 Sample Code Fragment to Illustrate the ASE Process

```c++
#include<iostream.h>
#include<conio.h>
int mul(int p,int q,int r);
int mul1(int p,int q);
int main()
{
    int s,w;
    cout<<"s="<<mul(2,3,5)<<"n";
    cout<<"w="<<mul1(3,2);
}
int mul(int p,int q,int r)
{
    int s;
    s=p*q*r;
}
    int mul1(int p,int q)
{
    int w;
    w=p*q;
}

In the above code, the derived and the recovery properties from the respected legacy program is shown in the Table 4.2.
Table 4.2 Retriving of Attributes from CPP to Java Conversion

<table>
<thead>
<tr>
<th>Legacy CPP</th>
<th>Derived Attributes</th>
<th>Recovered in Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>A1</td>
<td>✓</td>
</tr>
<tr>
<td>Methods</td>
<td>A2</td>
<td>✓</td>
</tr>
<tr>
<td>Statements</td>
<td>A3</td>
<td>✓</td>
</tr>
<tr>
<td>If….else</td>
<td>-</td>
<td>Not available</td>
</tr>
<tr>
<td>Switch</td>
<td>-</td>
<td>Not available</td>
</tr>
<tr>
<td>Operators</td>
<td>A4</td>
<td>✓</td>
</tr>
<tr>
<td>Data Structures</td>
<td>-</td>
<td>Not available</td>
</tr>
</tbody>
</table>

Table 4.2 gives the attribute satisfaction between CPP and Java. For a CPP code, ASE automatically retrieves all the Attributes that are named as A1, A2, A3 and A4. The different attributes identified in the Table 4.1 are Data, Methods, Statements, Conditional statements such as if…else, switch, Operators and Data structures used in the CPP program. Table 4.2 concludes that the retrieved attributes of CPP are migrated completely. In attribute retrieval module, the attributes are identified successfully and the interference removal module is the next module of our proposed ASE approach. Consider the following CPP legacy code from the above Table 4.1.

```c
int mul ( int p, int q, int r);
int mul1 ( int p, int q);
```

The above code undergoes some slicing algorithms and reengineering technique and produces an equivalent Java program which is given below.

### 4.5 RESULTS AND DISCUSSION OF ASE

The 50 programs used in the previous chapter were used to evaluate the efficacy of the proposed technique. Table 4.2 shows the performance parameters measured.
Table 4.3 Parameters Measured during Code Conversion using ASE

<table>
<thead>
<tr>
<th>Name of Program</th>
<th>Execution Time - ASE</th>
<th>Number of Errors- ASE</th>
<th>Translation Error- ASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>207</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>237</td>
<td>13</td>
<td>0.7</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>3</td>
<td>0.63</td>
</tr>
<tr>
<td>4</td>
<td>63</td>
<td>1</td>
<td>0.12</td>
</tr>
<tr>
<td>5</td>
<td>121</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>369</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>2.7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>589</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>339</td>
<td>115</td>
<td>4.64</td>
</tr>
<tr>
<td>10</td>
<td>147</td>
<td>2</td>
<td>0.18</td>
</tr>
<tr>
<td>11</td>
<td>34</td>
<td>47</td>
<td>8.88</td>
</tr>
<tr>
<td>12</td>
<td>208</td>
<td>16</td>
<td>0.94</td>
</tr>
<tr>
<td>13</td>
<td>480</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>89</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>163</td>
<td>7</td>
<td>0.46</td>
</tr>
<tr>
<td>16</td>
<td>245</td>
<td>180</td>
<td>9.17</td>
</tr>
<tr>
<td>17</td>
<td>86</td>
<td>12</td>
<td>1.42</td>
</tr>
<tr>
<td>18</td>
<td>545</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>4.1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>147</td>
<td>2</td>
<td>0.15</td>
</tr>
<tr>
<td>21</td>
<td>11</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>421</td>
<td>308</td>
<td>9.75</td>
</tr>
<tr>
<td>23</td>
<td>219</td>
<td>28</td>
<td>1.58</td>
</tr>
<tr>
<td>24</td>
<td>67</td>
<td>168</td>
<td>20.56</td>
</tr>
<tr>
<td>25</td>
<td>46</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>98</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>104</td>
<td>41</td>
<td>4.56</td>
</tr>
<tr>
<td>28</td>
<td>278</td>
<td>68</td>
<td>2.99</td>
</tr>
<tr>
<td>29</td>
<td>630</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>308</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>136</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>560</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>33</td>
<td>264</td>
<td>78</td>
<td>3.75</td>
</tr>
<tr>
<td>34</td>
<td>241</td>
<td>104</td>
<td>5.86</td>
</tr>
</tbody>
</table>
Table 4.3 (Continued)

<table>
<thead>
<tr>
<th>Name of Program</th>
<th>Execution Time - ASE</th>
<th>Number of Errors- ASE</th>
<th>Translation Error- ASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>81</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>36</td>
<td>181</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>37</td>
<td>228</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>38</td>
<td>60</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>39</td>
<td>158</td>
<td>17</td>
<td>1.43</td>
</tr>
<tr>
<td>40</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>41</td>
<td>321</td>
<td>16</td>
<td>0.65</td>
</tr>
<tr>
<td>42</td>
<td>382</td>
<td>4</td>
<td>0.15</td>
</tr>
<tr>
<td>43</td>
<td>146</td>
<td>31</td>
<td>2.69</td>
</tr>
<tr>
<td>44</td>
<td>364</td>
<td>22</td>
<td>0.86</td>
</tr>
<tr>
<td>45</td>
<td>19</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>46</td>
<td>34</td>
<td>18</td>
<td>3.52</td>
</tr>
<tr>
<td>47</td>
<td>219</td>
<td>64</td>
<td>3.48</td>
</tr>
<tr>
<td>48</td>
<td>3.8</td>
<td>45</td>
<td>15.41</td>
</tr>
<tr>
<td>49</td>
<td>187</td>
<td>17</td>
<td>1.19</td>
</tr>
<tr>
<td>50</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4.4 and 4.5 show the screenshots of the error generated using SRASG.

![Figure 4.4 Converted Java Program](image-url)

**Figure 4.4 Converted Java Program**
The outcome of the above Java program is evaluated as follows:

\[
\begin{align*}
p &= 2; & p &= 3; \\
q &= 4; & q &= 2; \\
r &= 5; & s &= p \times q \times r; \\
w &= p \times q;
\end{align*}
\]

When the above Java program is executed, it will display the error result as follows:

```
Figure 4.5 Execution of Java Program
```

To overcome the above errors, the second module of our work is implemented. Although the grammar is checked by the slicing algorithm there are some complicated tasks. When we analyze the migrating snippets, there are various issues and requires attention to further process. In the above transformed code, the variables \( p \) and \( q \) affect the value of \( s \) which leads to interference. In the legacy CPP the function overloading takes place to evaluate the result of \( s \). But the target system i.e., Java doesn’t support the overloading concepts and will affect the computation. It takes both the values resulting overlapping of each other by overlapping of these two values (for \( p \) 2, 3 and for \( q \) 2, 5) for a single variable. To overcome this error, the ASE method was applied. When the proposed ASE is applied to the above snippet, then the above code is modified as
\[
p=2; \quad q=3; \\
r=5; \quad s=p*q*r;
\]

After evaluating this snippet it evaluates other data as follows:

\[
P=3; \quad q=2; \\
w=p*q;
\]

By using our ASE the computation of the program is done by two separate modules. In migration, the modification takes place automatically by deleting the snippet \( q:=4 \) in the first module and vice versa. After the modification, ASE generates the model case for user satisfaction to avoid the abrupt migration. When the forthcoming code comes under positive case then it is ready for execution as the target system. We repeat the experiments for various programs. Table 4.2 gives the raw collection of the conducted experiments and the results are discussed below.

Another major difference between CPP code and Java code is that in code, the programmer is responsible for retaining any persistent object references and releasing all objects that are no longer necessary. In other words, the programmer is responsible for explicitly managing memory using the retain, release and auto release selectors. But in Java unused objects are automatically garbage collected and the programmer need not concern with the details of memory management. Once exception is created in a new thread in Java and that thread manipulates some object with CPP counterparts. In that case, we must explicitly create an auto release pool at the beginning of the thread. Java doesn’t use pointers, while CPP does. Therefore, we will need to change our code if the CPP method that are converting uses pointers. These are Java equivalents for all object pointers, but pointers such as \( \text{int}^* \) or \( \text{id}^* \) or \( \text{void}^* \) will not work. The CPP method must be changed to remove these references. CPP allows us to message null objects, but bridged Java does not.
Table 4.4 Code Fragment to Illustrate the ASE Process in Exception Handling

```c++
{
    Int sum=0, i=0, a[10]={0}, n=0;
    cin>>n;
    for(i=0; i<n; i++)
    {
        cin>>a[i];
    }
    for(i=0; i<n; i++)
    sum=sum+a[i];
    cout<<"sum=\"<<sum;
}
```

Table 4.5 Target Program After ASE

```c++
{
    int sum=0, i=0, a[]=new int[10], n=0;
    try
    {
        DataInputStream din= new DataInputStream(System.in);
        N=Integer.parseInt(din.readLine());
    } catch(Exception E)
    {
        System.out.println("Exception occurred"+E.getMessage());
    }
    for(i=0; i<n; i++)
    {
        try
        {
            DataInputStream din- new DataInputstream(System.in);
        } catch(Exception E)
        {
            System.out.println("Exception occurred"+E.getMessage());
        }
    }
    for(i=0; i<n; i++)
    sum=sum+a[i];
    System.out.println("sum=\"+sum);
}```
4.5.1 Stage 1 Results

Various CPP programs were taken with many attributes and constraints where the corresponding LOC of programs is estimated. The interferences are computed without ASE is denoted in the third column. Programs are numbered as 1, 2, 3 …50. After applying the ASE method, the interference is reduced in the migration system. The time required for evaluation of this process is given in the next column. Finally, the interference removal rate also computed for several programs is illustrated in Figure 4.7.

For the various programs from 1 to 50, the interference is computed with ASE and the above graph concludes that the ASE eradicates the interference compared to direct reengineering obtained from legacy system. The interference is very much reduced in the obtained output and hence it yields an improved new system. The added advantage of our work is after getting the result of the method, again gives this process as input to ASE.

Figure 4.6 Execution Time Analysis of ASE
While doing this recurrent process, it must be ensured that the peculiarities of the original system don’t modify in the target system. The Figure 4.6 gives the results of the recursive ASE process.

4.5.2 Stage 2 Results

When the interference and other abrupt errors are overcome by modifying the new system and so the efficiency and quality are also increased in the target system with ASE method which is discussed in Figure 4.7.

![Figure 4.7 Translation Error of ASE](image)

**Figure 4.7 Translation Error of ASE**

From the graph it can be concluded that the efficiency in code covers ion increased when the ASE is implemented along with SRASG. The Figure 4.8 concludes that ASE improves the efficiency of the migrated system.
This chapter discussed a methodology as how to reuse the components of the Legacy system effectively with reengineering technique. ASE works very well in all the environments. This ASE method identifies the interferences from the new system generated through the proposed SRASG. It also identifies the syntactic errors from a program whether a new system is a programming language. The Java Bridge allows us to convert our application or framework at the speed (Marko Bajec et al. 2006) that makes sense of our situation, testing and debugging every step of the way. By the ASE the following features of CPP are converted to Java such as Exception handling, Function template, Function overloading, Call by reference etc., The output consists of structured Java code that performs the same function.

This approach is further enhanced by embedding the method directly to the reengineering process. The experimental results conclude that the ASE is correct, effective and suited for high level, complicated large scale systems.