State Provisioning with XAP

Wei Wang
Peihuan Zhong
Master Thesis

State Provisioning with XAP

Peihuan Zhong
Wei Wang

31st of May 2006

Computer Science Department
Växjö University

Supervisor:
Jesper Andersson
Abstract

This master’s thesis intention is to develop a Java language framework with XAP providing state provisioning function. The idea is that the state from an application should be transferred between different application instances opening up for continuous application use. By using our framework prototype, applications which are provided by XAP can implement the state provisioning function. We first introduce our framework and its core mechanisms. To be able to verify it, we also present an evaluation of the feasibility and practicability of our framework.

Key words: Java, JVM, State Provisioning.

Acknowledgements

First, we would like to thank our supervisor Jesper Andersson for his continual support and direction through the project. We also like to thank Marcus Edvinsson for his patient help and guidance on XAP. Last, we would also like to thank our friends and families for all the support throughout our studies.
CONTENTS:

1 Introduction ............................................................................................................ 8
  1.1 Main Concepts and Work Flow ............................................................................ 8
  1.2 Problem Statement .............................................................................................. 9
  1.3 Research Questions ............................................................................................ 10
  1.4 Restrictions ........................................................................................................ 10
  1.5 Research Method ............................................................................................... 10
  1.6 Thesis Outline .................................................................................................... 11

2 Background ........................................................................................................... 12
  2.1 State migration of Java ...................................................................................... 12
  2.2 The XAP System ............................................................................................... 12
    2.2.1 What is Provisioning? ................................................................................... 13
    2.2.2 Workflow of XAP ....................................................................................... 13
    2.2.3 Platform Architecture ............................................................................... 14
    2.2.4 The Ranking Framework and the Composition Framework .................... 14
    2.2.5 Application Families ................................................................................ 15
  2.3 The Inject/J ---source code transformation tool .............................................. 15

3 Solution of State Migration .................................................................................. 17
  3.1 Concepts ............................................................................................................. 17
  3.2 Overview of the Approach ................................................................................ 17
  3.3 Saving and Restoring the Data State ................................................................. 18
  3.4 Saving and Resuming the Execution State ...................................................... 18
  3.5 The Details of Our Jump Mechanism ............................................................... 20
  3.6 Checkpoint Accessing into the Method ............................................................ 23
  3.7 Transmitting the State Information .................................................................. 25

4 The Persistent Framework .................................................................................. 26
  4.1 The Transferable Class ..................................................................................... 26
  4.2 The Data Container Class ................................................................................ 26
  4.3 The Checkpoint Class ....................................................................................... 27
  4.4 The Checksaver Class ...................................................................................... 28
  4.5 The MigrationEvent Class .............................................................................. 28
  4.6 The State Provisioning Class .......................................................................... 28
  4.7 The Design Pattern of the Framework ............................................................ 28

5 State Provisioning Function of XAP ................................................................. 31
Contents of Figures:

FIGURE 1-1: HIGH LEVEL WORK FLOW OF THE APPLICATION PROVISIONING SERVER (APS) .......................... 9
FIGURE 2-1: WORK FLOW OF APPLICATION PROVISION SERVER (APS) .................................................. 13
FIGURE 2-2: ARCHITECTURE OVERVIEW OF THE XAP PLATFORM ..................................................... 14
FIGURE 2-3: ARCHITECTURE OF THE INJECT/J TOOL ........................................................................... 15
FIGURE 2-4: META-MODEL OF INJECT/J ............................................................................................... 16
FIGURE 3-1: CODE SAMPLE OF TRANSFORMATION OF SEQUENCE STRUCTURE ........................................ 20
FIGURE 3-2: CODE SAMPLE OF TRANSFORMATION OF SELECTION STRUCTURE ...................................... 21
FIGURE 3-3: CODE SAMPLE OF TRANSFORMATION OF LOOP STRUCTURE ............................................... 21
FIGURE 3-4: TRANSFORMATION FROM FOR-STATEMENT TO DO-WHILE STATEMENT ............................... 22
FIGURE 3-5: TRANSFORMATION FROM WHO-STATEMENT TO DO-WHILE STATEMENT .............................. 22
FIGURE 4-1: THE DATAContainer CLASS .............................................................................................. 27
FIGURE 4-2: THE CHECKPOINT CLASS .................................................................................................. 27
FIGURE 4-3: THE CHECKSAVER CLASS .................................................................................................. 28
FIGURE 4-4: THE STATEProvisioning CLASS .......................................................................................... 30
FIGURE 5-1: WORKING PROCESS OF THE STATE PROVISIONING FUNCTION (SAVE THE STATE) .......... 32
FIGURE 5-2: WORKING PROCESS OF THE STATE PROVISIONING FUNCTION (READ THE STATE) ............. 33
FIGURE 6-1: THE SCREEN-SHOT OF THE ECLIPSE PLUG-IN ................................................................. 34
FIGURE 6-2: PSEUDO CODE OF ADDING CHECKPOINT BY INJECT/J ..................................................... 36
FIGURE 7-1: TIC-TAC-TOE EXAMPLE .................................................................................................. 37
FIGURE 7-2: FILE STRUCTURE OF TIC-TAC-TOE .................................................................................. 38
FIGURE 7-3: THE SCREEN-SHOT OF THE TWO VERSIONS OF TIC-TAC-TOE GAME AFTER MIGRATION .. 38
FIGURE 7-4: THE SCREEN SHOT OF MIGRATION OF FIBONACCI .......................................................... 41
FIGURE 8-1: UNFOLDING DUPLICATION TRANSFORMATION FOR LOOP STRUCTURE .......................... 44
List of Tables:

TABLE 4-1: API OF THE STATEPROVISIONING CLASS ................................................................. 29
TABLE 5-1: THE TCPCLIENT API ................................................................................................. 32
TABLE 7-1: EXECUTION PERFORMANCE COMPARISON ................................................................. 39
TABLE 7-2: GROWTH IN BYTE CODE SIZE OF PROGRAMS .............................................................. 40
TABLE 7-3: EXECUTION MEMORY CONSUMPTION COMPARISON ................................................. 40
1 Introduction

Recently, we have been witnessing a significant growth in the number of Java-enabled mobile devices. With those new mobile devices we can download and run software, which brings up many new opportunities. A variety of mobile applications are offered in the industry. In the near future mobile devices and in-car computers will have the same capabilities as stationary computers. The differences between these systems will disappear; and users will focus on the key aspect - software applications. A prominent requirement is to implement seamless integration with transparent handover between hosting devices. Provisioning should be provided for continuous use of a specific application. For instance, users can continue to work in the same settings after interruption, by resuming the state of the applications rather than loading the saved files before the interruption.

Due to the motivation above, we carry out this thesis work by extending the XAP [1] system with the function of state provisioning. In this section, for the sake of helping readers to get an overview of our work, we introduce the main concepts and work flow at the very beginning. Afterwards, the problem statement, research questions, restrictions and research method of our project are presented briefly. Then the outline of the thesis is given in the end of this section.

1.1 Main Concepts and Work Flow

To start the thesis, we would like to explain the concept of ‘state provisioning’ with an example. Johan, an office staff, is working in his office, using a software application on a desktop computer. For some reasons, he needs to move to another room to continue the work. And he needs to move frequently in his daily work. Instead of carrying a mobile device with him all the day, Johan simply activates a new application instance on the new desktop computer or mobile device at the new work place. And the state of the application on the new computer is exactly the same as that on the computer in his office when he left. This kind of resuming states of software application at different places is called ‘state provisioning’.

eXtended Application Provisioning (XAP) is able to accomplish the tasks of application provisioning via customize and configure the provisioned applications before retrieval. To extend XAP with state migration we must handle the data state of a given application and execution state. The data state of an application consists of two kinds of classes: the first one is device-independent classes, which are the same for the specific application on different machines. The second one is device-dependent classes, which vary from one machine to another.

To fulfill this demand, we try to extend the XAP system with state provisioning capabilities, and provide a framework and a checkpoint mechanism to allow seamless application handover between devices. The framework that we designed for programmers to develop applications with state provisioning function is named SP_Framework. It enables state transferring between devices. Programmers may easily perform functions of saving and loading states on XAP servers by merely invoking the methods provided in the SP_Framework. To save and load states, we develop a network module named SP_Server, which may easily be integrated in the XAP providing helper functionality related to communication.
The state provisioning that we propose for the XAP system can be described by the workflow in figure 1.1: a user downloads the customized application from XAP, and configures it with device-dependent classes according to the specification (hardware information, platform information and so on) of the device. When the user starts a new application instance on a different computer, the existing application has to shut down, although the work is uncompleted. The state of the existing application will be uploaded/saved on the server. After that the user downloads the application to the current device, if the application is not installed to this device yet, and loads the state, which was just uploaded on the server.

1.2 Problem Statement

One of the problems to provide state provisioning for application with XAP is how to build a persistent framework for the application developers. What we need is a generic, plug-able and extendable framework, on which programmers can develop the transferable applications. At the same time the state migration details should be hidden from the developers. In fact not all the classes need to be transferred during the migration, for instance, classes that contain algorithms. Therefore, we categorize the classes in an application by their attributes when states are migrated. In this framework we only consider the classes that we need to transfer. We call this kind of classes Transferable Classes.

Another problem in state migration is the execution state of objects. Although Java supports capturing the state of all the objects with the serialization mechanism, saving the
execution state in Java is not an easy job. Java is an interpreted language and requires an interpreter (the Java Virtual Machine, or VM for short) to execute. The second problem is how to access the method call stack and program counter information, and how to restore it. Java does not allow access to JVM on the source-code level. Some systems choose to modify the JVM to capture the execution state of the application. For example, JVM is extended to be DJVM in [18]. Although this is an effective and efficient way, it is not portable. The devices, which XAP provides applications to, are sometimes mobile devices, for instance, Smart Phones. They have a pre-installed JVM and in most cases users are not willing to or maybe prohibited to modify the system software on the mobile devices. Some systems use byte code instrumentation, which does not modify the JVM. For instance, a transformation tool on byte code level is developed in [7]. It is merely suitable for Mobile Agents [7]. The applications provided by XAP are just regular applications and are quite different for various Mobile Agents. Thus, we need a mechanism to capture the execution state of classes on the source-code level.

Last but not least, we need to extend the XAP server with facilities to upload and download states, since state provisioning is a totally new function for XAP system. We need to build a server-side service to handle the states from different applications.

1.3 Research Questions

The questions which will be discussed in this thesis are concerned with transferring the states of applications. This may relate to the solutions for capturing and resuming states and requirements for application development. To clarify our research goals, we sum up our research questions as

- How to develop a persistent framework to implement the strong migration for the applications provided by XAP?
- How should programmers develop the applications which are able to use the state provisioning function of XAP?

1.4 Restrictions

When we started to study the migration of application states, we realized that this was a huge topic and too complex task for a thesis work. To make our thesis feasible and practicable, we must restrict our research field. Therefore handling of multi-threaded applications are not in the focus of the framework development. Besides, although we believe that the difference between mobile device and desktop computer will be resolved in the future, we focus on desktop computers at this moment. Another restriction is we only focus on the applications which may be provided by XAP since there are thousands or even millions of applications in this productive software realm. Some may be very complicated and impossible to resolve. This is far beyond the reach of a thesis work. These delimitations help us focus on the topics of feasibility and efficiency of the framework.

1.5 Research Method

In response to the questions presented above, we started with conducting a literature study of the related topics. As to state migration, we have studied previous works that investigate this problem. The theories and techniques used for mobile agents can help to
give us some inspirations for our persistent framework. In order to study the problems practically and validate our solution, we implemented the persistent framework based on the application, Tic-Tac-Toe, which was used as an example in a thesis about XAP [1]. The state of Tic-Tac-Toe is migrated from a GUI version to a Text version using our framework. In the process of developing the persistent framework for Tic-Tac-Toe, we endeavored to find the key points of answering the research questions. After that, we generalized the framework to make it available for common applications. We provide some supporting tools to help programmers to develop software with state transferring function and to release them from the burden of understanding the details of the framework.

1.6 Thesis Outline

This thesis is organized as follow: In chapter 2, we introduce some background knowledge including XAP, Inject/J and State migration of Java. Chapter 3 is devoted to the theory of our migration mechanism. We describe migration for the data state, execution state and also our jump mechanism with checkpoints. In Chapter 4, we show the persistent Framework we have developed. Then we describe the state provisioning function of XAP in Chapter 5 and state migration developer support in chapter 6. After the evaluation of the results in chapter 7, we give a comparison between our approaches and related work in chapter 8 and a discussion in chapter 9. Finally, the conclusion and future work are given in Chapter 10.
2 Background

Before we introduce our state-provisioning prototype, we present some related technologies. Areas covered are: state migration, the XAP system, and source-to-source transformations with Inject/J.

2.1 State migration of Java

State migration is widely used in the area of mobile agents. However, our State Provisioning function is not intended for mobile agents. So we define the concept in our own way. Hereby state migration means continuous execution of an application at another machine by transferring the execution state from one machine to another and resuming computation. From a technical point of view, the state of a Java application consists of three kinds of data:

- **Code state**: it refers to the byte code of the Java applications.
- **Data state**: The data state includes internal state information and external state information. The internal state information contains the data information of the running applications, e.g., basic attributes of the applications, local variables and etc. The external state information contains information of other applications and the files that the application is using, for instance a database.
- **Execution state**: also called the thread state or runtime state. It indicates the execution position of the current thread. It contains the control information for the application running, the program counter, the stack and etc.

Application migration can be divided into two groups depending on whether it transfers the runtime state or not: Weak Migration and Strong Migration [3]. Weak Migration, also called non-Transparent Migration, is when the application requests migration, only the data state is saved and transferred to the destination computer together with the application code. On the destination computer the application can not be resumed at the same code position as it was before the migration. Instead, the application starts from the program entrance. The program has to be executed again to the old code position. Strong Migration, also called Transparent Migration, means: when the application requests migration, both the runtime state and the data state have to be saved and transferred to the destination computer together with the program code. On the destination computer the application can resume in exactly the same state and at the same code position as it was before migration. Both mechanisms require capturing the state information and a reestablishment of the saved state during restart.

2.2 The XAP System

The XAP system, which stands for eXtended Application Provisioning (XAP) system, developed by School of Mathematic and Systems Engineering at Växjö University. It is intended for automatic customization and configuration. The applications to provision are all custom-made applications. The customization and configuration of applications can be comprehended to include or exclude the different features or traits of applications due to restrictive hardware specifications. According to the capability of the target device, XAP as a provisioning server can filtrate the content or components of the applications
that are unsuitable for that device. For instance, if the device has a monochrome or black and white screen, offering the user of this device a color picture is unavailing. XAP accomplish the task of application provisioning, however its main job is to customize and configure the provisioned applications. In order to extend a standard provisioning framework, we should dig into the storage of the bundle repository and the logics for displaying to the client what is available, and also what time applications are retrieved from the bundle repository. [1]

2.2.1 What is Provisioning?

Provisioning is a basic and significant concept, which should be clarified before we go to details. Based on the definition of the JSR-124 [17], the term provisioning refers to the activities of advertising services to a client device, and to process requests and deliver the service to the device. In other words, provisioning, on the server side, includes the operation of presenting what can be transferred to the client side, and responding to the request of clients and distributing the required items. The physical form of client service is referred as client bundle, or just a bundle. It may be an archive file containing Java byte code and the resources/data it uses. [JSR-124]

2.2.2 Workflow of XAP

As mentioned before, the purpose of the XAP system, on one hand, is to allow users to configure the application before it is downloaded; on the other hand, it automatically removes the application features that the target device does not support, because of the limited hardware specifications, and customizes or tailors the features of applications for the users. From Figure 2-1, we can see that the XAP conceals all operations from the user and uses the Web Interface for interaction with the user. [1]

The Web Interface, which is incorporated in the system, provides a Graphical User Interface (GUI) and works as a front-end between the XAP composition platform and the user. This Web Interface is a set of dynamically generated HTML pages, which acquire their content from the Provisioning Framework in the XAP. All those HTML pages are on a Tomcat web server, which works as Java Servlet and Java Server Pages (JSP) container. While the presentations of the provisioned applications are implemented with JSP, the business logics of provisioning behind this Web Interface are abstracted in
Actions or Java Servlets. Action is a concept class in the Struts web application framework. [1]

The XAP configurable composition platform consists of a number of frameworks is the major component in the system framework. And the entire element framework can be instantiated when creating a composition environment. Based on the target runtime platform and preferences/request from users, the composition platform generates a configured and customized application. It supports many processes like components selection, composition support and storage facility. The Platform Architecture is presented below. [1]

2.2.3 Platform Architecture

At point 1 the process is initialized when the Ranking Framework receives a hardware and software specification of the target platform for a specific application. Then the Ranking Framework asks for descriptions of the appropriate applications from the Application Repository at point 2. Based on the previously received target platform specifications, we need to adapt the application description, modify it, and remove those that do not match. At point 3, the modified application descriptions are transferred to the Composition Framework. After that, the Composition Framework receives a configuration profile at point 4. This profile is to specify which application features should be included or excluded when the application is composed. Finally, the Composition Framework delivers the composed and configured applications in form of Configuration Profile and application description. [1]

2.2.4 The Ranking Framework and the Composition Framework

Since the ranking framework and the composition framework are two core parts of XAP, we would like to give a brief description of these.

The ranking framework [1] is used to rank objects according to certain criteria. After ranking, every ranked object is given a score or marks, which indicates the degree it satisfies the criteria.

The composition framework [1] assembles the components according to the hardware environment of a specific target. What it does is not just simply adding all selected
components into a package. To some complex programs, they need adaptation, modification of source code, compilation and so on.

2.2.5 Application Families

All the applications that the XAP system can provide are stored in a special repository. The largest units in an XAP repository are named Application Families, consisting of a number of applications with similar features. The architecture of each application is stored in a tree structure, named Application Feature tree (AF tree), which starts with an Application Root Node. The child nodes of the Application Root Node indicate components of this application and can be marked as optional, mandatory or alternative to signify whether the components are indispensable or elective. In addition, we could also store auxiliary information by attaching them to the nodes. [1],[2]

2.3 The Inject/J source code transformation tool

In our approach, source-to-source transformation takes the code written by the developers as input and generates new code that fits our framework. What our transformation tool does is to change the structure of the original code and add some methods and method-invocations. The original codes and generated codes are both compilable by Java compiler thus compatible with the JVM. From the functionality point of view, the difference between them is that original code is executable but not state-transferable, while the generated code is both executable and state-transferable.

Inject/J is an open source tool written in Java, which can perform source to source transformation [8]. It is based on the Recoder/Java [8] transformation library and can be used to transform Java source code. Figure 6.2 shows the architecture of Inject/J.

![Figure 2-3: Architecture of the Inject/J tool.](source: Inject/J Software Transformation Language Specification 2003)

It consists of a meta-model for automatic source code transformation, a script language to describe the transformation operations, a script processor which interprets the transformation operations that are to be performed and a meta-programming library (RECODER) for the eventual program transformations. [8],[13]
The Inject/J software transformation language is a script language describing and specifying complex transformation operations according to the model entities and conceptual operations [8]. The meta-model of Inject/J consists of two layers, one is the upper layer which defines a conceptual view, the other is lower layer that defines both the language specific mapping between the conceptual entities and semantics of analysis operations and also the semantics of transformation operations based on the primary and secondary code-level transformation. The upper layer is responsible for design level transformation. It is used to hide the complicated syntax tree of the underlying languages brought in by the lower layer. Inject/J software transformation language is a script language that provides full access to the upper layer and offers language constructs which navigates throughout the meta-model, selects model entities and accomplishes the transformations of model.
3 Solution of State Migration

Before introducing our solution, we introduce some relevant concepts that we used in the research work. Based on those concepts we give an overview of our approach. Then we sketch our schema to approach the state migration. Hereby we address issues like: how do we save and restore the Data State and the Execution State? After that we move on to the details of the mechanism that we used in source-code level migration. Finally, we give some explanations on how to access checkpoints in methods.

3.1 Concepts

- **Direct Transferable Class and Indirect Transferable Class**: The application provided by the XAP consists of two kinds of classes. The first one is device-independent classes, which keep exactly the same for applications on various kinds of target systems. The other one is device-dependent classes that may be different from one target system to another. The class, which defines objects that may be transferred, is referred to as a Transferable Class. The category of Transferable Class may be further divided into two categories: Direct Transferable Class, which refers to the device-independent classes, and Indirect Transferable Class, which refers to the device-dependent classes or some Classes whose state can not be transferred directly.

- **Checkpoint**: it is used to record a position of the execution. Every statement is assigned a serial number, which is used to track the position. And the checkpoint is introduced to hold the information of the current serial number. In the process of resuming execution, we can jump to the line of the code, that the checkpoint points to.

- **Level**: It is introduced to record the depth of the statement inside compound statements or recursive methods. Every level has it own checkpoint value. Thus, we can keep track of checkpoints in a hierarchical structure. This will record the program position accurately.

- **Data Container**: In the process of transferring state, we need a carrier for the state information. For this reason we design a Data Container to carry all the necessary information such as checkpoint, the objects states and so on.

3.2 Overview of the Approach

We have already stated that application state comprises three parts: **the code state, the execution state and the data state**. If we could find a solution for each of these three parts, we can unquestionably complete our task under our current delimitations. Since the XAP system already has the function of provisioning the code to a client device, transferring the code state to the client will not be a problem. In the rest of this chapter, we will only focus on how to transfer **the execution state and the data state**.

The Java language has many useful capabilities, which help us to implement the migration of data state, e.g., object serialization [19] that allows Java objects to be mobile or persistent and Remote Method Invocation (RMI) [20]. However, Java does not provide any capabilities for the persistence of control flows (threads). In other word, the execution state of a Java program is only accessible inside the JVM.
There are many ways to transfer the execution states. In all of these methods the execution states can only be transferred between two identical applications. In our situation, XAP can provide different versions of applications to the users according to different hardware configurations. So the state provisioning function of XAP should have the ability to handle the migration between two application instances which may not be identical.

In order to solve this problem, we try to develop a persistency framework, which should be a generic, plug-able and extendable framework and be capable of helping programmers to develop transferable applications.

Considering the Direct Transferable Class, the application code is the same on all kinds of devices after downloaded. So we can transfer both the data state and execution state to another device. As to the Indirect Transferable Class, we just transfer the specific information related to its states to other devices using object serialization. When another device downloads the state of the device-dependent classes, it will execute the code again according to the saved state.

As to the execution state of the main thread, there are already some systems that are able to capture and restore the state in Java. However, they modify the JVM. Considering the portability, we present a checkpoint mechanism, which is introduced on the source-code level and does not modify the JVM.

### 3.3 Saving and Restoring the Data State

If we transfer and resume the complete Data State, we can meet the requirements of Weak Migration. Saving and restoring the Data State, which means saving all the local variables and objects of the application, is the most important step in state migration process. Our solution to transfer all variables and objects of device-independent classes is straightforward. We just put them in a Data Container and retrieve values from the Data Container when resuming. As to the objects of device-dependent classes, the implementations of the different versions are quite different and sometimes incompatible. For example, in the case of Tic-Tac-Toe, the implementation of the Output Class of GUI Version is totally different from the one of Text Version. In Java, it is impossible to assign an object to different classes, which is not of the “same” type, e.g. inheritance. Therefore it is impossible to assign an object of the Output Class of the GUI Version to the object of the Output Class of Text Version. So we encourage the developers to use a separate state object to hold state information. In addition, developers are also required to implement a method to reset the object according to this unique state. We believe that it would not be a big burden for the programmers to fulfill this requirement.

### 3.4 Saving and Resuming the Execution State

As mentioned in Chapter 1, in the problem statement, the migration for execution states of classes is a difficult job that requires a lot of work. Java, as an interpreted language, requires an interpreter to execute Java programs. Java execution state information is located in the interpreter. Besides, Java does not allow any access to the JVM internals on the source-code level. In this way, some approaches to the state transferring problem are based on modifying the JVM, such as Bouchenak’s [4]. If we choose to modify the JVM to enable execution state capturing of applications, it might be an effective solution. However, we will sacrifice the portability of common State Provisioning Framework.
And this is not acceptable, since XAP mostly provides applications for mobile devices and where it is impossible or difficult to modify the VM. Also, the approach with byte code instrumentation [21] is not suitable for applications provided by XAP. The possibility of migrating states with a Byte code instrumentation approach is based on the fact that the programs that require state-migration have exactly the same byte code. However, the applications provided by XAP may be different in different environments, e.g., hardware specifications. For example, the Text Version of Tic-Tac-Toe has a different Output Class comparing to GUI Version. Definitely, the byte codes of applications are kind of diverse.

Based on the discussion above, we know that capturing the execution state information is a tricky problem. Instead of all the other approaches mentioned above, we propose an alternative mechanism called Checkpoint Mechanism. This checkpoint mechanism is generally used to record the execution position on the source code level. With the help of checkpoints, we can jump to the position where the program was executing when migration was initiated. When discussing a jump mechanism, we have to mention the goto statement in the C language and Assembly Language. Since goto statement is not a good construct in most cases [22], Java does not permit the goto statements. Nevertheless, in our case, it would be a simple and effective solution for jumping to a certain position in the code. Unfortunately, the substitutes of goto in Java, break and continue, only permit escaping from a block statement. No alternative way to jump into a compound statement is presented in Java [6]. In one of the proposed solutions [5], if-statements are used to steer clear of the code executed before the “current position”. However, the number of the if-statements introduced by this method is high. Thus, it is not an ideal solution for jumping. We propose a straightforward and efficient solution. The two key points of our approach are switch-case statements and a multi-level checkpoint technique. This reduces the overhead; and the main thread can resume to any position of the body of main function.

The switch-case statement is used to get the position from the checkpoint and a certain label, which is used to tag the position of the checkpoint. In this way, we simulate the ‘Goto-statement’ on the top-level. In order to go to a lower level, we introduce the multi-level checkpoint, which is composed of an array of bytes. It is unreasonable for a method to have more than 128 statements on one level. Thus a byte type is sufficient. The indices of the array indicate the level while the values of array elements indicate the position of the code. The Multi-Level Checkpoint technique enables application execution to be resumed in the middle of a compound statement, such as if statement, for statements. The sub-statements after the “resume-point” and the subsequent statements that should be executed until the control flow reaches the top level are duplicated on the top level.

After capturing the state information at the checkpoint, the next task is to save the data state in the execution position. To do so, firstly, we have to make sure that no further code of the application can be executed after the method--saveState ()--is invoked. This can be automatically done by the exception/error handling mechanism provided by the language. We apply this strategy to save the current data state and execution state. The approach is as follow:
The method, which initiates the process of saving state, throws an exception named MigrationEvent.

An encapsulating try-catch statement is inserted for each method that might initiate the state saving.

The body of the method is put in a try block and the methods for saving the state are located in the catch block.

3.5 The Details of Our Jump Mechanism

In this section we explain details of our mechanisms. For a better understanding of our mechanism we define the concept of Structure. A Structure is a basic unit of programming logic. This has much in common with the concept of Basic block [23] which is widely used and accepted. However, there is a big difference that the Basic block does not include any jumps or jump targets whereas Structure does. Our mechanism is based on the three basic structures:

- Sequence Structure
- Selection Structure
- Loop Structure

Every Java method is based on these three structures. This means that if we can deal with these three basic structures, we can handle almost all Java programs. According to the characteristics of the three structures, different strategies must be applied for each structure.

First, we focus on the easiest one, Sequence Structure. Within a sequence structure, you perform an action or event, and after that the next action is executed. A sequence can contain any number of statements. But there is no chance to branch off or skip any statements. So we just need to keep track of which action or event the program is running at and then the system may jump to the position directly in the process of resumptions. The code transformation sample is illustrated in Figure 3-1.

![Figure 3-1: Code Sample of transformation of Sequence Structure.](image)

With the checkpoint and switch-case statement mechanism, the sequence structure is implemented. Now we can consider the Selection Structure. In this structure, the program
works as: first you ask a yes-or-no question, and then depending on the answer, you take one of two courses of actions. Some people prefer calling the selection structure an if-then-else structure because it fits the statement. This structure is to some extent more complicated than the Sequence Structure above, since the checkpoint and switch-statement is based on the Sequence Structure. The code transformation sample is illustrated in Figure 3-2. At this moment this is only suitable for the if-else statement. As to the switch statement, we do not need to add the checkpoint outside the switch statement anymore, because our checkpoint mechanism is based on switch statement. However the code will be blocked after the keyword ‘case’ of the switch-statements in the original source code. They also need to be transformed as a Selection Structure.

![Figure 3-2: Code Sample of transformation of Selection Structure](image)

The last structure is the Loop Structure, which needs to be restructured in a similar way as adding checkpoints to the Selection Structure. The code transformation sample is illustrated in Figure 2.3.

![Figure 3-3: Code Sample of transformation of Loop Structure](image)
As to the while-statement and for-statement which are other types of loop structure, we need to transform them to do-while statements first and then add the checkpoints. They are illustrated in Figure 3.4 and Figure 3.5 respectively.

![Figure 3-4: Transformation from for-statement to do-while statement.](image)

```plaintext
for(s1; s2; s3)
{
    statement 0;
    statement 1;
    :
    statement n;
}
```

```plaintext
if(s2){
do{
    statement 0;
    statement 1;
    :
    statement n;
    s3;
}while(s2);
}
```

![Figure 3-5: Transformation from while-statement to do-while statement.](image)

```plaintext
while(condition ){
    statement 0;
    statement 1;
    :
    statement n;
}
```

```plaintext
if(condition ){
do{
    statement 0;
    statement 1;
    :
    statement n;
}while(condition );
}
```

For compound statements, we use a scheme that every time the execution position is reached, the \textit{Level}, which indicates the index of checkpoint array, increases; when the compound statement is finished, the index decreases. Hence the position of each level is kept in the checkpoint. Another problem we have to pay attention to is if the statements contain a break-statement, we should add the label at the beginning of the corresponding loop, because the switch-case statement also supports the break-statement. If we just add the break-statement in the switch-case statement directly; the system will behave incorrect semantically and treat the break statement as the break from the switch-statement rather than the original loop.
3.6 Checkpoint Accessing into the Method

In order to access the method and add the checkpoint, we force every method to use the same checkpoint instance. In order to do this, we configure a byte array. In the array, the value of the index indicates the level and the value of each element holds the position of the corresponding level. Although there are many checkpoint instances, they share the same data, since the byte array is static. Thus, when accessing methods, the level will increase by one and the checkpoint will start to count. Nevertheless, some problems emerge when accessing into the checkpoint. The first problem is we refer to as **Identical Level Confusion**.

The Identical Level Confusion is a side effect of the checkpoint mechanism. Consider the following assignment statement:

```
Variable=function1 () + function2 ();
```

If there are two methods accessible in the same single statement, the confusion will occur. Suppose the checkpoint is entered into function1 and function2, the number of levels will be the same to both of the functions. If we assume that the execution position will reach inside function2. When resuming, the system will misunderstand the position, because they have the same level and the execution reaches function1 before function2. To handle this problem, we split the expression. A single expression, which contains more than one method, will be decomposed into a sequence of atomic operations. The above statement is decomposed as follows.

```
Variable=function1 ();
Variable=Variable+function2 ();
```

This technique helps us to separate two functions on the same level and guarantees that we avoid the Identical Level Confusion.

The second problem comes with the return-statement in the middle of a method. If the return-statement is at the bottom of the implementation, the `checkpoint.levelExit()` method (the checkpoint class are described in section 4.3) will be carried out before the return. And the level will return to the former one before it enters the methods. However, if the return-statement is in the middle of the method and is embedded in several levels, the `checkpoint.levelExit()` is not able to deal with this situation. A solution to this problem is to introduce another array, which holds the information of the levels before the system enters the method. If there is a return-statement inside a number of embedded statements, we will add the `checkpoint.levelExit(int i)` right before the return statement. When this return statement is going to be executed, the `checkpoint.levelExit(int i)` will change the checkpoint back to the former level and then the system executes the return-statement.

When we access to the methods, we have to consider whether it is necessary to add checkpoints inside. If we add the checkpoint in the function, it will increase the overhead. And it may be possible that the computation of the overhead is several times more than the computation of the method itself. In some situations, some methods should not be accessible; otherwise side effects will arise. For example, the method in our prototype `menuInterface()` has two parts, one is to display the menu, the other is to get the list of
users. Suppose we have added the checkpoint to the method and the execution position (status of the program) is waiting for the user input from the keyboard. We may successfully capture the state and resume it on another machine. However, the application will start with continuing waiting for the input of the keyboard and nothing will be shown on the screen to inform the user. This is because the state of the screen cannot be captured in our framework and the states of Swing objects are not transferable.

After pondering the case presented above, we come up with the concept of *Atomic Method*, which does not allow the addition of checkpoints. Our framework will ignore atomic methods while putting the checkpoint into the code of application. The framework will regard methods with only one statement as Atomic Method automatically in order to reduce unnecessary overhead. To declare an Atomic Method is quite simple. The developers simply need to add “_atom” as the suffix to method names.

If a Method is declared as an Atomic method, we need to change the strategy of saving the state. This is because there is no checkpoint added inside the method. The position only stays outside of the method. If the data states are modified inside the method and state capturing is activated in the process of executing the current method, there will be a problem when resuming. This is because the data states are not the states of corresponding checkpoint. To demonstrate the problem practically, we present an example below.

```java
Method_atom(){
    Code 1;
    If (Object_something1==null){
        Object_something1=Object_something2;
    }
    else {
        Object_something1=null;
    }
    Code 2;
    Code 3;
}
```

There is no checkpoint added inside the method. In the if-statement of the method, we assume that before executing the method, the value of the Object_something1 is not null. After the if-statement, the value of Object_something1 will be changed to null. We assume that when the program reaches Code 2, state capturing is activated. When the program is resumed later, the method will be re-executed from Code 1, because there is no checkpoint inside and it is not finished when the program saves the state. After resuming, the Object_something1 is assigned with Object_something2. In fact Object_something1 should be null.

For this reason we save the state before the body of the method starts to run and insert a flag inside the Atom to notify the system that it does not need to save the data states again. Before leaving the method, the flag will set to false to notify the system that it requests to save the data states when the saving state is activated.
3.7 Transmitting the State Information

Once the state is captured, the state information will be serialized to a file and transferred to the XAP server via TCP. In the process of resumption, the file containing the state information will be retrieved via TCP from the server and then desterilized to be state information. The detailed process will be presented in Chapter 5.
4 The Persistent Framework

The persistence framework is generic, plug-able and, extendable. It helps programmers to develop transferable applications. The class, which should be transferred, is defined as Transferable Class. In the process of transferring the state information, the internal state would be carried by a Data Container. The Data Container also contains the information of the valid checkpoint, which holds the execution position of the main thread. For the device-dependent classes, the implementation of some methods would be different. So we only record the execution state on method-level. The requirements or restrictions for this persistent framework are twofold. On one hand, all the methods that are used in the main function should be defined in the framework; on the other hand method invocation is always allowed inside the main function.

4.1 The Transferable Class

Since we divide the classes of applications into device-independent classes and device-dependent classes, we inevitably adopt distinct transformation strategies for them. Considering the device-independent classes, the code is the same in all kinds of devices after downloading. We can transfer both the data state and execution state to another device without adverting anything. As to the device-dependent classes, the code may be different for distinct devices. Some devices, for instance, support graphical interface, while others may merely support a textual interface. The code is different for these two kind devices. In this situation, we just transfer the certain information related to its states to the target device using object serialization. When another device downloads the state of the device-dependent classes, it will execute the code again according to the saved state.

Among all the classes, we define the device-independent classes to be Direct Transferable Class. The device-dependent classes and the classes whose state are not able to be transferred directly are defined as Indirect Transferable Class.

4.2 The Data Container Class

The Data Container mentioned in chapter 3 works as a carrier to hold the internal state information. It is designed to carry all the necessary information such as checkpoint, objects of Transferable Classes and so on. The implementation of the DataContainer class is quite simple: It uses a Hashtable to store the objects of programmer-defined classes and Checksaver class, which is used to hold the state of checkpoint class. It implements the Serializable interface that ensures the classes that implement this interface will have their state serialized or deserialized.
4.3 The Checkpoint Class

The *checkpoint* class consists of a bunch of methods for manipulating the checkpoint operations. Among all the member functions, there are three key methods, namely *moveNext()* , *levelEntry()* and *levelExit()* . The *moveNext()* is used to move the position of the checkpoint to the next position. The *levelEntry()* usually comes with the *levelExit()* in pairs. They are used for compound statements. The *levelEntry()* is used when the program enters a deeper level and the *levelExit()* is used when existing this level. Also checkpoint has two important fields for storing the position and tracking the level. The first one is a byte array used to store the position of every level. The other one is an integer used to keep track of the index of the array, that is to keep track of the level. Every object of this class can access *moveNext()* , *levelEntry()* and *levelExit()* because they are static. Therefore, the checkpoint is possible to access every public member functions of every class.
4.4 The Checksaver Class

Since most of the members of the checkpoint Class are static, it is not suitable to serialize them. In this case we introduce Checksaver. It has similar data structure as the checkpoint class but the data members are not static. It has some member functions which are used to aid assigning the values between the Checkpoint and Checksaver classes.

4.5 The MigrationEvent Class

When we save the state, we make use of the exception/error handling mechanism to pause the program. Before we save the state, we throw a MigrationEvent exception and then the program stops its execution to jump to the catch statement and save the state. The migrationEvent extends the Exception class.

4.6 The State Provisioning Class

The StateProvisioning class is the core class in the persistent framework. It extends the DataContainer class and implements several methods like setStateName(), saveState() etc. All these make it possible to handle the operations when migrating the states. The applications simply invoke the methods in this class to save the state, read the state and resume, and migrate states.

4.7 The Design Pattern of the Framework

To best use our framework, we recommend that programmers satisfy several criteria when they program the class that will be included in the main function:

- The classes that have the same behavior but different features (Indirect Transferable Class) should have the same class name and same interfaces.
- Defining Indirect Transferable Classes inside Direct Transferable Classes are not recommended.
- The body of the main function should be the same in all versions. For instance, the Text version of Tic-Tac-Toe and GUI version should have the same main function basically.

The first criterion ensures that different versions have nearly the same implementation in the class, which contains the main function of the application. The second requirement is a restriction. In the process of transferring, Direct Transferable Classes will be assigned...
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setInsideAtom(boolean enterTrue_exitFalse)</td>
<td>Tell the program that it is inside an Atom Method or not at present.</td>
</tr>
<tr>
<td>setResume(resume_or_not)</td>
<td>Set the mode (resume or normal)</td>
</tr>
<tr>
<td>setUserId(user_id)</td>
<td>Set the user name for logon.</td>
</tr>
<tr>
<td>setPassword(password)</td>
<td>Set the password for logon.</td>
</tr>
<tr>
<td>setStateName(state_name)</td>
<td>Set the name of the state.</td>
</tr>
<tr>
<td>setSelectStateNum(selected_num)</td>
<td>Select which state to read if there is more than one state to read.</td>
</tr>
<tr>
<td>setOverWrite(mode)</td>
<td>Choose to overwrite the state or not if the state name already exists.</td>
</tr>
<tr>
<td>activateSave()</td>
<td>Throw a migrationEvent exception and stop current execution.</td>
</tr>
<tr>
<td>saveState()</td>
<td>Save the state.</td>
</tr>
<tr>
<td>readState()</td>
<td>Read the state.</td>
</tr>
<tr>
<td>selectMode()</td>
<td>Show the selection of the mode and let the users select the mode of the application (resume or normal)</td>
</tr>
<tr>
<td>getDataCotainer()</td>
<td>Get the Data Container.</td>
</tr>
<tr>
<td>resume()</td>
<td>Resume the state.</td>
</tr>
</tbody>
</table>

Table 4-1: API of the StateProvisioning class.

back to themselves directly. When an object is serialized, all objects invoked by it will be serialized at the same time. If Direct Transferable Class uses the Indirect Transferable Classes, they will be assigned back directly. So the programmers should keep in mind that, if a Direct Transferable Class uses the Indirect Transferable Classes, it is an Indirect Transferable Class from then on. The third one makes it possible to migrate between different versions of applications. Different versions of applications should be treated differently on the level of the implementation of Indirect Transferable Class. However, on the level of using the interface of the Indirect Transferable Class, different versions must be keep identical. The successful migration between different versions is based on this common ground.

In our state-provisioning framework, we implement some methods to help programmers make state-transferable applications. In Table 4-2, we list all the methods implemented. In order to guide the programmers to customize the process of state migration, we also have some methods to be overridden by programmers. In Table 4-2, we list all methods. Programmers are encouraged to use the state-provisioning framework API when they override these methods.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>showModeSelection()</td>
<td>Show the user the selection of the application. For example, (1: Run from the very beginning 2: Resume the application from the existing state.)</td>
</tr>
<tr>
<td>inputUserNameAndPwd()</td>
<td>Let the user to input the user name and password.</td>
</tr>
<tr>
<td>multiStateHandler()</td>
<td>If there is more than one state, this method is invoked.</td>
</tr>
<tr>
<td>identicalStateNameHandler()</td>
<td>If the state name already exists in the database, this method is invoked.</td>
</tr>
<tr>
<td>saveDetector()</td>
<td>It tells the application when to invoke the saveState() and what to do after save the state.</td>
</tr>
<tr>
<td>buildDataContainer()</td>
<td>It puts the state data into Data Container. It will be generated by the source-translation tools.</td>
</tr>
<tr>
<td>resumeProcessor()</td>
<td>It gets the state data stored inside Data Container and assign them back to the corresponding object to resume the application. It will be generated by the source-translation tools.</td>
</tr>
<tr>
<td>inputSaveProcessor</td>
<td>Before saving, it is used to ask the user to input some information, such as username, password and the name of the state.</td>
</tr>
</tbody>
</table>
5 State Provisioning Function of XAP

As we have mentioned in the problem statement, we need to extend the XAP server with support for uploading and downloading the states. This can be done by developing a server-side service which can handle the states from different applications.

5.1 State Provisioning Function of XAP

XAP can really be used for this but for the simple prototype a quick implementation outside XAP is more efficient. Since XAP already has the function for users to download the tailored software, the migration of the source code will not be our consideration. To fulfill the requirements for State Provisioning, we need a platform for storing and downloading the state information. XAP is really able to be used for this. However, implementing the prototype outside XAP is simpler and more efficient at the beginning. So, we use a Server and Client module to solve this.

The main communication between server and clients are querying, modifying state data, downloading and uploading the state data. Considering the modification of the database, our initial scheme is to use Apache Axis as the Web Services Server. Apache Axis can provide services like modifying the database and uploading the Hash table of state data to the server. However, if the Hash table contains some custom objects, it is inconvenient for Axis to pass it. The programmers have to register both the Serializer and Deserializer of the user-defined objects. Moreover the authentication of the Web Services is still a problem.

Another means of communication is Transfer Control Protocol (TCP). TCP enables two hosts to establish a connection and exchange streams of data. In our scheme the ObjectInputStream and ObjectOutputStream are adopted to transfer the information. However, to transfer the custom objects with TCP, the custom class should be recognized by the server. Acknowledging the server in every custom class would be bothersome for programmers. And it is unnecessary for the server to recognize the custom classes. The custom classes have to be recognize by the target device rather than the server which only stores and transfers the states. As to RMI, the problem is similar. The local interfaces of remote objects have to be defined in the server. So our solution is to write the state data into a file, save it in the local device and then send it to the server by TCP. In this way, the server does not need to know what kind of custom objects included in files. In order to support many users to build the communication with the server simultaneously, every user uses one thread to communicate with the server.

Because all the applications that XAP system can provide to users are stored in a special repository, we expect that state data can also be stored in another special repository. But some related information of the state data, e.g., the path of the state, are not appropriate to be stored with the state data in the repository, therefore we put them in a database.
5.2 The TcpClient API

Our module adopts a two-tier software architecture, namely Client-Server architecture. It is implemented with two major classes TcpServer and TcpClient. The TcpServer is located on the server side; it will listen to all the communication requests and then create threads to build the connections. The TcpClient class located on the client side; it offers Application Programming Interface for programmers to communicate with the server. The TcpClient API is presented in Table 1.

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>TcpClient(user_id, password, client_path)</td>
</tr>
<tr>
<td>Boolean</td>
<td>checkAuthentication()</td>
</tr>
<tr>
<td>Boolean</td>
<td>checkStateName(state_name, software_id)</td>
</tr>
<tr>
<td>Boolean</td>
<td>sendState(Software_id, state_name, Cotainer, overWrite)</td>
</tr>
<tr>
<td>Hash table</td>
<td>readState(selected_state_num)</td>
</tr>
<tr>
<td>String[]</td>
<td>getStateList(software_id)</td>
</tr>
</tbody>
</table>

Table 5-1: the TcpClient API

For the Constructor, the user_id and password will be used in authentication process. The client_path refers to where the programmer stores the temporary file that holds the state data. The checkAuthentication() is used for the programmers to check whether the user name and password are valid. The sendState(Software_id, state_name, Cotainer,
over_write) and readState(selected_state_num) methods are imposed to send and read the state information from the server. ThegetStateList(software_id) can help the user to get the list of the states existing in the server. All these methods make the storage of state information and resumption of state straightforward for application developers. The details of the process are shielded from users of the application. The whole process and workflow of the communication is illustrated in Figure 5-1 and Figure 5-2.

Figure 5-2: Working Process of the State Provisioning Function (Read the State).
6 State migration Developer Support
In order to assist developers and hide the core of the State Provisioning Framework as much as possible, we have developed some supporting tools.

6.1 Developer support for Eclipse
Our approach of realizing the state transferable application is quite simple. However, if it done by hand, it would require a lot of work and developers are easy to get lost. So we develop an Eclipse Plug-in to direct the developers to build their own state transferable applications and generate some of the necessary code fragments automatically. Figure 6.1 is the screen-shot of our plug-in. In our framework, there are two classes, one is Direct Transferable Class, and the other is Indirect Transferable Class. And we have one special class that contains the main function. With our plug-in, the programmer can create a class by selecting which one they want to create. And then the Eclipse will generate the code automatically.

![State Provisioning Class Creation](image)

Figure 6-1: The Screen-Shot of the Eclipse plug-in.

6.2 Source to Source transformation—Inject/J
Also, Inject/J is utilized to develop a source to source transformation tool. We mainly this supporting tool to do two things. One is to generate the methods to save the data states, the other is to add the checkpoint. In the following subsection, we describe the brief mechanism of the transformation tool developed by Inject/J.
6.2.1 Generating the Methods to Save the Data States

First, we use Inject/J [8] to elevate Local Variable Declarations inside the main function. Before elevating Local Variable Declarations, we need to transform the for-loop statements to while statements, since there may be variable Declarations in the for-loop statements. In our approach, there are two key methods in MainClass in the process of capturing and resuming the data state: buildDataContainer() and resumeProcessor(). The first one is to save all the objects, local variables and the checkpoint information to the Data Container another one is to assign all the information inside existing Data Container back to the objects, local variables and the checkpoint. In order to reduce the burden of the developers, we use Inject/J to generate these two methods automatically. However, for the sake of accessing the object and local variable declarations, we need to elevate them to attributes of the Main Class and set them to static. The reason is as follows: in our framework there is only one class, namely MainClass, which contains the main function for every application. Most framework API is defined in this class. And the framework only supports single thread at present. So we can set the method, saveState(), to be static, because there is only one thread running all the time. In this way, every class may access this method to save the state information of the current thread. The method, saveState(), invokes buildDataContainer() to save all the objects. buildDataContainer() is also static. Therefore, all the variables and objects that are saved inside buildDataContainer() should be static.

In the process of elevating variable declarations, Inject/J identifies all the local variables in the main function and then copy them to the outside of the main function and delete the original declarations. After that, we generate the body of buildDataContainer() using the API, addData(objectID, object). We use the names of variables or the names of objects as the objectID in order to avoid name collision. In the process of generating resumeProcessor(), we have to add the type casting automatically since getData(objectID) only returns an object. Inject/J conducts the whole process of transforming the source code.

6.2.2 Adding the Checkpoint

To help understanding our strategies of adding checkpoints with Inject/J, we have to clarify two concepts: Checkpoint Code Block and List. Checkpoint Code Block indicates the code block between two ‘cases’ in the switch statements. List refers to one of the types of Inject/J, which means a collection of some objects of certain types. We wrap the process of adding the checkpoint as a function. Parameters are taken as List of Checkpoint Code Block. In the implementation of functions, we use recursion in order to keep the code as simple as possible. The pseudo code is illustrated in Figure 6.2.

The Inject/J script for our framework is given in the Appendix A. It elevates all the variables and objects declarations inside the main function to the outside of the function body and add corresponding statements into two methods buildDataContainer() and resumeProcessor() automatically to save the data state. And then it adds the checkpoint to the main function. When we add the checkpoint, we define a library, splib, which contains a method, addCheckpoint(). But in our current script, we can not access the compound statement, thus we can not complete it. We will discuss it in Chapter 9.
```java
addCheckpoint(Code_Block_List)
if(Code_Block_List.size==1 & & is not a return statement)
return;

alreadyExit=0;
Add: "switch(checkpoint leve1Entry())" in front of the body of switch statement.

Integer Case_Number=0;

for each codeBlock in Code_Block_List do{
  Add the "case"+Case_Number+1+";" +";
  in front of the Checkpoint Code Block

//Selection Structure
if(isBranch){
  1. Transform the code and generate a new List (its name is New_CBL)
     of Checkpoint Block from each branch of the Selection Structure.
  2. addCheckpoint(New_CBL)
}
//Loop Structure
else if(isLoop){
  1. Access into the loop to get each statement and generate a new
     List (its name is New_CBL) of Checkpoint Block.
  2. addCheckpoint(New_CBL)
}
else {
  if is Return and it is not the last statement of the method body:
    1. calculate the number(num) of level it needs to exit
    2. add "checkpoint levelExit(num);" before it
}
else if(Case_Number = Code_Block_List.size-1){
  Add "checkpoint moveNext();" after the Checkpoint Code Block
}
}
Add "checkpoint levelExit();".
}
```

Figure 6-2: Pseudo Code of Adding Checkpoint by InjectJ.
7 Evaluation
In this Chapter we will evaluate the techniques as implemented in the framework. First, we will use a small application to test the feasibility of our framework. Then we evaluate the overhead of time and space penalty introduced by the source code transformation.

7.1 A Simple Experiment
The program we used in the experiment is a very popular game, Tic-Tac-Toe, which already exists in XAP system repository as a demo application. The game exists in two versions: graphic and text. XAP is able to configure and customize it before download. The configuration profile criteria is introduced in the XAP [1]. We can use a repository editor [1], a supporting tool, to simply configure the configuration profile. The Configuration profile of Tic-Tac-Toe is included in Appendix B.

In this section we first demonstrate the feasibility of our state provisioning framework, providing the XAP with state provisioning function. And then we discuss some implementation issues and solutions.

To have a good grasp of the experiment, we first clarify the architecture of the Tic-Tac-Toe application. As shown in Figure 7.1, it is a very simple architecture consisting of four main components, a Game Engine, a Computer Player, an Input Interface and an Output Interface. The component, Output Interface, has various features that one can choose when customizing the application. This is completed by the XAP.

![Figure 7-1: Tic-Tac-Toe Example.](image)

In this example, only the Output Interface is the device-dependent class. And it is impossible to migrate all the execution state and data state from the graphic version to text version, because their mechanisms and implementation are very different. As a consequence we choose to use weak migration and execute the class from the beginning to a certain stage according to the data state. It extends the Indirect Transferable Class and has two features; one is text version while the other is graphic version. Game Engine holds all the state of the game, for instances, which grids are taken, whose turn and so on. The rest of the text version and the graphic version of the game engine are identical. Therefore it is Direct Transferable Class. The other two interfaces, Computer Player and Input Interface, they do not have any states and are also identical. Therefore they are not Transferable Classes.
The two critical methods that need to be implemented in Indirect Transferable Classes are:

- public Object returnShareData();
- public void resume();

The first method returns the data which is part of the state of the Indirect Transferable Class. The `resume()` will inform the Indirect Transferable Class how to execute the class again according to the state data in the DataContainer. Nonetheless, the data alone, which the method `returnShareData()` returns, might be insufficient to resume the executing state, since this class may use other classes during its execution. Thus, for a convenient resumption, we furnish the whole DataContainer to the programmers by setting methods that are used to get the data out to be static. In this case, the Output Interface needs some information about the output stage, for example, to show the menu or output the game board. Besides, it also needs the Game Engine to show the users the state of the game board.

![Figure 7-2: File Structure of Tic-Tac-Toe](image)

Among all the files of the game, there is a class named MainClass. It implements the main function and all the functions (Listed in Table 4-2) which needed to be overridden.

![Figure 7-3: the Screen-Shot of the two versions of Tic-Tac-Toe game after migration](image)

### 7.2 Evaluation of Overhead

Since our mechanism for the migration of execution state is source code transformation with checkpoints inject code, the extra time and space expenditure are inevitably. In order to evaluate the overhead and get an insight into the performance, we benchmark it by executing several Java applications. From the comparison between the original programs and the transformed programs, we have learned valuable and significant lessons about the potential defects and limitations of state migration on top of our State Provisioning Framework.
Our evaluation comprises two steps. First we compare the performance to validate the practicability. Second, we implement the migration of a single program using our framework to testify the feasibility. The evaluation programs we choose to use are Fibonacci, Quick-Sort, Queen-N, Richards and DeltaBlue. Here is a short description of all these programs.

### 7.2.1 Descriptions of Test Applications

Fibonacci is a famous recursive and extremely inefficient algorithm for calculating Fibonacci numbers. [9]

“Quick-Sort, a well-known sorting algorithm, is significantly faster in practice than other \( \Theta (n \log n) \) algorithms. Quick-Sort is a comparison sort and, in efficient implementations, this algorithms is not stable.” [10]

The N-Queens benchmark solves the combinatorial hard chess problem of placing \( N \) queens on an \( N \times N \) chessboard such that no queen can attack any other. It uses a recursive search for a placement of the queens that meets the correct conditions. [11]

“Richards is an interesting medium-sized language benchmark (400-500 lines). It simulates the task dispatcher in the kernel of an operating system. There are several versions for Richards. The version we used for evaluation is ‘Richards gibbons’”. [12]

“DeltaBlue is about 1000 lines of Java code. It has also been translated into many languages including C, C++ and Smalltalk. The algorithm is described in "The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver" by Bjorn N.” [12]

Conducting the experimental evaluation includes three parts: (1) Time Penalty, (2) File Space Penalty and (3) Memory Penalty. when executing the benchmark, no migration takes place.

### 7.2.2 Time Penalty

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Elapsed Time (ms)</th>
<th>Original</th>
<th>Transformed</th>
<th>Growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci(40)</td>
<td></td>
<td>3193</td>
<td>9804</td>
<td>207.0%</td>
</tr>
<tr>
<td>Quick-Sort(1500000)</td>
<td></td>
<td>1518</td>
<td>1940</td>
<td>27.80%</td>
</tr>
<tr>
<td>Queen-N(13)</td>
<td></td>
<td>1390</td>
<td>1643</td>
<td>18.20%</td>
</tr>
<tr>
<td>Richards()</td>
<td></td>
<td>4</td>
<td>8</td>
<td>100.0%</td>
</tr>
<tr>
<td>DeltaBlue(10000,10000)</td>
<td></td>
<td>615.6</td>
<td>645.4</td>
<td>4.841%</td>
</tr>
</tbody>
</table>

Operating System: Microsoft Windows XP Professional Service Pack 2 5.1 Build 2600
CPU: Intel Pentium M Dothan Processor(1.73 GHz)
Physical Memory: M523440 KB
JDK: jdk1.5.0_06

Table 7-1 : Execution Performance Comparison.

Entering and exiting the level of the checkpoint will reduce the performance substantially, since every time the system leaves one level, it does not only decrease the level but also reset the checkpoint of the level to zero. The result demonstrates this potential penalty. The time penalty of Fibonacci and Richards are extremely high, this is because a large number of recursions occurs in the process of execution of these two tests. As to other three tests, performance is much better due to less amount of recursion. In Table 7-1, we
can study the result of the time penalty test.

### 7.2.3 File Space Penalty

The file space penalty of our framework consists of two parts. One is our framework library, which is fixed. The other one comes from the checkpoints and related functions that are added to manipulate checkpoints. This is quite different from various structures and implantations of the programs. In our evaluation, we only compare the latter file space penalty. The results are shown in Table 7-2. We can see the results are approximately similar to the fact that the rate of file space penalty increases in an inverse proportion to the file size. The overhead of the smallest program, Fibonacci, is quite high, while the largest program, DeltaBlue, is quite low. The reason is that the structures and implementations of large programs are usually complicated. And many methods will be considered to be Atomic Methods automatically. For instance, DeltaBlue, it consists of many methods. Some of them have only one statement. Our schema to add the checkpoint skips methods which have only one statement.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Byte Code Size (bytes)</th>
<th>Original</th>
<th>transformed</th>
<th>growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci(40)</td>
<td></td>
<td>391</td>
<td>636</td>
<td>62.7%</td>
</tr>
<tr>
<td>Quick-Sort(1500000)</td>
<td></td>
<td>1410</td>
<td>1980</td>
<td>40.4%</td>
</tr>
<tr>
<td>Queen-N(13)</td>
<td></td>
<td>1080</td>
<td>1610</td>
<td>49.1%</td>
</tr>
<tr>
<td>Richards()</td>
<td></td>
<td>993</td>
<td>1230</td>
<td>23.97%</td>
</tr>
<tr>
<td>DeltaBlue(10000,10000)</td>
<td></td>
<td>2410</td>
<td>2790</td>
<td>15.77%</td>
</tr>
</tbody>
</table>

The same runtime environment as above.

Table 7-2: Growth in Byte Code Size of Programs.

### 7.2.4 Runtime Memory Penalty

Based on the result, our approach nearly not give any extra memory consumption. The reason is that the memory penalty from the checkpoint is mainly derived from the array of bytes which is used to hold the checkpoint. And the number for any byte array member is normally smaller than 1000 for most applications. We can ignore the memory penalty since it only consumes around 1000 bytes in the results.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Memory Consumed (Byte)</th>
<th>Original</th>
<th>transformed</th>
<th>growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci(40)</td>
<td></td>
<td>139632</td>
<td>140816</td>
<td>0.848%</td>
</tr>
<tr>
<td>Quick-Sort(1500000)</td>
<td></td>
<td>12144576</td>
<td>12145760</td>
<td>0.010%</td>
</tr>
<tr>
<td>Queen-N(13)</td>
<td></td>
<td>144552</td>
<td>145744</td>
<td>0.825%</td>
</tr>
<tr>
<td>Richards()</td>
<td></td>
<td>218824</td>
<td>219736</td>
<td>0.417%</td>
</tr>
<tr>
<td>DeltaBlue(10000,10000)</td>
<td></td>
<td>218464</td>
<td>219512</td>
<td>0.480%</td>
</tr>
</tbody>
</table>

The same runtime environment as above.

Table 7-3: Execution Memory Consumption Comparison.
7.2.5 Migration of Application Containing Recursive Methods

In our experiment, Tic-Tac-Toe, there exists no recursive method. We can not test the practicability of applications that have recursive methods by this simple experiment. This forces us to build another experiment test for applications with recursive methods. In order to simplify the process and make a straightforward understanding for the readers, we pick up the famous recursion Fibonacci program as the test case. We adjust the structure of the Fibonacci a little bit for displaying the calculating process. Each time when the thread enters the recursive method of Fibonacci, the program will ask the user if they want to continue the calculation or save the state and output the parameter N, where N is an integer as the Fibonacci Numbers argument.

![Figure 7-4: The Screen Shot of Migration of Fibonacci.](image)

The screenshot of the Migration can be seen in Figure 7-2. The state of Fibonacci application is captured and resumed flawlessly. This demonstrates that the checkpoint mechanism is also suitable for some recursive methods. However, our framework is not able to resolve all recursive methods. The reason for this will be discussion in Chapter 9.5.
8 Related Work

In this Chapter we will compare our work with another three techniques of transparent migration of the Java programs. Two of them are also based on source-code transformation, while one is based on byte code instrumentation.

8.1 Comparison of Approaches

Fünfrocken [5] has implemented a thread serialization mechanism on the source code level. However, instead of simulating the Goto statement with ‘switch’ statements and unfolding duplication technique, it introduces a large number of if-statements to skip “already executed code”. Hence the overhead increases greatly. Tatsuro et al. [6] adopted an approach that introduces the switch statement and unfolding duplication technique. However, it extends the Java Language and hence it increases the programmers’ burden. Also, portability is another consideration that rules out this approach for our work.

Some researchers use byte code instrumentation. The Execution Model for Agents [7] presents the thread serialization mechanism using byte code rewriting. Compared with the two source-code transformation approaches presented above, this technique comes with several advantages. First, it is almost impossible to capture the entire execution state of a running thread with source code transformation. Inspecting the values that are on the operand stack of the currently executing method seems unattainable, for instance. Second, since a more precise control is offered at the byte-code level, a byte-code instrumented thread serialization mechanism has superior efficiency with respect to space and time overhead. For example, Fünfrocken [5] a worst-case byte code inflatable factor of 470%, while byte-code instrumentation meet the worst-case factor of only 107% for a parallel program. The distinction in efficiency is due to the low-level byte code instructions which are very suitable to manipulate the control flow in a program. Byte-code, for example, can overlap the already executed code in the process of reestablishment to avoid re-execution by means of a simple Goto instruction. However, this Goto instruction is unavailable on the source-code level. Besides, the simulation of Goto statements on source-code level costs much. In [5], a large number of if-statements are introduced to skip of already executed code. This mechanism can be optimized using unfolding duplication technique [6]. Nonetheless, the reformative efficiency with byte code transformations is still unacceptable. Last but not least, we can have much more flexibility with byte code transformation since it allows inserting reestablishment code before the execution of the default super-call within a constructor, whereas this is forbidden at the source-code level.

All the reasons above imply that it is a wise solution of transparent migration for two identical applications. However, in our situation, the applications provided by XAP may be two slightly different instances. This is because the applications will be configured according to the platform or hardware of the target devices before being downloaded. In our approach, we implement the checkpoint mechanism on the level over the implementation of device-dependent classes. Identical interfaces and source-code of the device-dependent classes achieve an identical implementation of the main thread and all the non-atomic methods. It transfers the state of the device-independent and re-executes the device-dependent class in the target device. Thus we could hide the “untransferable”
part from the checkpoint mechanism.

The Fünfrocken [5] and Java Extension [6] approaches are both intended for mobile agents. There are several differences between these two solutions and ours. Firstly, their migration is based on methods. The purpose of their migration is for pervasive computation, so they are used to distribute certain computation to other machines. Our intention is to transfer the state of the whole application so that users can work in a continuous environment. Secondly, their solutions also transfer the code to machines. With regards to our approach, the code state is provided by the Application Provisioning system and the code would be reconfigured by XAP before being downloaded. The applications that will transfer state are different in our setting, and their solutions are not feasible for applications provided by XAP. Thirdly, when transferring and capturing the state, is decided by the programmers. However, for a normal application, it is not a wise idea to capture the state after every execution, since it would make the application very slow. Our State Provisioning Framework provides the programmers to develop the applications that when the state capturing can be decided by the user. In our solution, capturing the state only occurs users desire to. For example, in our prototype, we add the method of saving state in the action listener of the windows. Once the user try to close the windows, the method of saving state will be activated.

8.2 Comparison of Overhead

The technique for adding the checkpoints to the compound statements are quite different. Fünfrocken definitely increases the overhead quite a lot due to the large number of if-statements imported by this scheme. The file space penalty is around 400% in its evaluation testing, Fibonacci. Even though the unfolding duplication technique [6] optimizes this, it is still unacceptable. Unfolding duplication actually extract the deeper level statement of the compound statement out. So every method only has one level. But this will also increase the file penalty a lot. Our initial scheme is to use the unfolding duplication technique of Java. As a result we predict that it will introduce a great overhead due to the duplication. In general, the code size grows in proportion to the square of the depth of the loops. In their evaluation, the file space penalty is normally more than 140%. Since we use multi-level checkpoint to solve the problem of compound statements, our file space penalty is less than Fünfrocken [5], the Java Extension [6] and even the Byte Code Transformation Approach [7]. However, we sacrifice some performance due to frequent entering and exiting levels. In the performance benchmark, we also use the same programs as our evaluation programs. And the results show that the performance we lost is quite similar.

Our framework works very well with the programs that have less recursive methods. As to normal applications, the users usually save the state after certain operations. Also, some operations are so fast to terminate that it is impossible for the users to save their application in the process of the operations. As a result, we expect that many methods are Atomic Methods, and are not necessary to checkpoint every statement. Hence, our framework is able to get the similar performance to DeltaBlue or even better.
Figure 8-1: Unfolding Duplication Transformation for Loop Structure.
9 Discussion
In this section we discuss relevant issues that relate to the thesis and also the implementation status of our prototype.

9.1 The Maximum Depth of the Checkpoint
In our scheme, an array of byte is used to hold the checkpoint for all the levels (depth). In the Java language, there is a maximum size for the Java heap space. This induces a maximum number of the members to the array of bytes. The maximum depth for the Fibonacci evaluation test is approximately between 61800000 and 61900000 in the hardware environment given in the evaluation. If more, it would raise an out of memory error in the console. We believe that for regular scale application, 1000 levels are quite enough. Due to the limitation of time and experience, we can not predict that whether it would be a problem for some complex applications with numerous recursive methods. Since complex applications means noticeably heap space and it is difficult for the programmers to take the array of checkpoint into account. Nevertheless, the risk is acceptable for most situations.

9.2 Exception of Direct Transferable Class
In our approach, besides the states of the objects of device-dependent classes, some other states are also impossible to transfer directly at present. This kind of classes have to be defined to be Indirect Transferable Class, despite its implementation is the same on different machine, such as the class of the layout. Although the state is transferred from GUI Version to GUI Version or from Text Version to Text Version, the state of the screen is still missing. So we have to rerun certain member functions to output the information again. The programmers should keep in mind that the classes of these kinds shall be defined as Indirect Transferable Classes..

9.3 Inject/J limitation
We have used the 0.81 version of Inject/J for this thesis project. Some interfaces are still not implemented, such as the `getLoopStatements`, `getBranch(integer i)` of the `StatementFragments` and so on. For this reason the access to if-else-statement are blocked. We can not find any solution to compound statement also. Due to the limitation of Inject/J we do not successfully achieve our muti-level checkpoint automatically by the source code transformation and implement the methods to transform the for-loop statement to while-statement. We, however, believe that the implementations of these interfaces will come out with the later version of Inject/J. Then we can complete the unfinished work.

9.4 Multi-Thread and External state information
When considering the solution of multi-threaded applications and external state information in our State Provisioning Framework, we immediately realized that this would be a rather complicated task. A lot of situations are necessary to be investigated. In order to make the prototype feasible, we only stay within the domain of single-threaded applications and internal state information transfer.
9.5 Local Variables in Recursive Methods

Although the experimentation of Fibonacci was successful, it does not mean the framework works for all recursive methods. In our approach, the local variables of the recursive method are only partily transported. It means the local variables in the stack are missing. In our experiment of Fibonacci, there is only one local variable and with the help of Multi-level Checkpoint, according to the current value of the that variable, it is able to calculate the values of the variables in the stack before saving the state again. Nevertheless, for some situation, this is impossible. This problem may possibly be solved by a detector. If an recursive method is detected, the framework will declare an array of objects to hold variables in the stack. Nevertheless this is the possible solution, as to a reliazation, it is scheduled as future work, since recursive method is always able to be implementd in an non-recursive way and the problems that we would confront would be many and difficult.

9.6 Restriction of Source-Code-Level Transformation

Due to the source code level transformation, some restrictions are brought in naturally. These restrictions are similar to other migration schemes based on source code level. First it is impossible for the state migration to take place in three places: in a class initializer (a static initializer), in an instance initializer and in a constructor. It is impossible to resume a class initializer, instance initializer or a constructor since they are invoked by the run-time system while a class is loaded or an object is created.
10 Conclusion and Future Work

In this thesis, we have presented a persistent framework, which allows for transparent migration of applications provided by XAP and a server-side service for handling a state provisioning function for XAP. In our approach, the Object Serialization of Java is utilized to manage the data states, Multi-Level Checkpoint is introduced to resolve the execution state as well. So, the two research questions are answered at the same time (see Chapter 3 to 5). In order to validate the functionality and estimate the overhead, we carried out an evaluation ending with a promising result. To release the burden of programmers we developed a very simple Eclipse plug-in which can generate some requisite code and a source-to-source transformation tool that reconstructs the source code to reliaze the persistent framework automatically. However, due to the limitation of Inject/J, the transformation tool is not completely finished.

In the process of creating the Tic-Tac-Toe prototype, we came across many problems. Those problems helped us to solve many practical difficulties in the persistent framework. Meanwhile, in the evaluation process, the five benchmarks also expose many bugs and defects of the framework. This helped us to improve to our framework. For instance, we initially utilized an array of integer to hold the checkpoint of every level; and brought in another array of integers to hold the information for each corresponding former level. However, this approach increases the file space penalty, time penalty and runtime memory penalty substantially. We then reconsider the solution and revolve to use byte array to hold the checkpoint instead of integers.

The core technique of our state migration scheme is the Multi-Level Checkpoint mechanism. Compared to the Java Extension Scheme [7], our approach increases the time penalty slightly but it pays a high price in terms of the file space penalty.

Due to the limitation of Inject/J, we could not finish an Instrument Tool for automatically checkpoint adding. However, we believe that this will be implemented without much effort after the later version of Inject/J comes out.

Considering the exception of device-independent class (see Chapter 9.2), our future work will focus on resolving this problem in order to liberate programmers from handling those tricky problems. For example, we will extend the framework to realize mapping the windows layout information automatically. As a result, the information of the screen would be a Direct Transferable Class in the future during the migration.

The second step is to move on to explore how to extend the framework to support multi-thread applications and external state transferring. After that, the framework may be used to provide state provisioning function for most XAP applications.

In order to achieve a more generalized State Provisioning framework, an implementation for storing the local variables on the stack in recursive methods is in the schedule for future work as well.
Reference


[3] Nishkam Ravi, Cristian Borcea, Liviu Iftode, Split Smart Messages: Middleware for Pervasive Computing on Smart Phones, Department of Computer Science, Rutgers University, USA.


[5] Stefan Fünfrocken. Transparent Migration of Java-Based Mobile Agents: Capturing and Re-establishing the State of Java Programs


Appendix A

This appendix provides the Inject/j script for our framework.

StateProvisioning.ij :
uses splib;
script StateProvisioning {
    foreach c in class ('user_defined.MainClass') do {

        //Elevate the variables declaration out of
        foreach m in c.methods do {
            if (m.fullName.equals("user_defined.MainClass.main") ){

                //get all the declarations inside the main body out
                //and delete the original ones
                if(m.getDeclarations("*", false).size>0){
                    foreach d in m.getDeclarations("*", false) do {
                        resFragment = ${}$;
                        resFragment.append("static ");
                        resFragment.append(d.getStaticType().Name);
                        resFragment.append(" ");
                        resFragment.append(d.getDeclarationText());
                        resFragment.append(";");
                        c.addToMembers(resFragment);
                    }
                }
        }

        //modify the buildDataContainer() and the resumeProcessor()
        foreach a in c.attributes do {
            attrName=a.Name;
            typeName=a.getStaticType().Name;
            foreach buildcontainer in c.methods do{
                if(buildcontainer.name.equals("buildDataContainer")){
                    code="StateProvisioning.addData("\"\"
                        +attrName+\"\",\"+attrName+\")\";

                    //insert the code to buildDataContainer
                    resFragment1 = ${}$;
                    resFragment1.append(code);
                    buildcontainer.afterEntry(resFragment1);
                }
            }
        }

        //insert the code to resumeProcessor()
        foreach resumeprocessor in c.methods do{

    }
if(resumeprocessor.name.equals("resumeProcessor")){
    code2="";
    //generate the code to add to resumeProcessor()
    if(typeName.equals("int")){
        code2=a.Name+"=Integer.parseInt(
            StateProvisioning.getData("\n"+
            attrName+"\n").toString());";
    }
    else if(typeName.equals("double")){
        code2=a.Name+"=Double.parseDouble(
            StateProvisioning.getData("\n"+
            attrName+"\n").toString());";
    }
    else if(typeName.equals("float")){
        code2=a.Name+"=Float.parseFloat(
            StateProvisioning.getData("\n"+
            attrName+"\n").toString());";
    }
    else if(typeName.equals("boolean")){
        code2=a.Name+"=Boolean.parseBoolean(
            StateProvisioning.getData("\n"+
            attrName+"\n").toString());";
    }
    else if(typeName.equals("long")){
        code2=a.Name+"=Long.parseLong(
            StateProvisioning.getData("\n"+
            attrName+"\n").toString());";
    }
    else if(typeName.equals("short")){
        code2=a.Name+"=Short.parseShort(
            StateProvisioning.getData("\n"+
            attrName+"\n").toString());";
    }
    else if(typeName.equals("byte")){
        code2=a.Name+"=Byte.parseByte(
            StateProvisioning.getData("\n"+
            attrName+"\n").toString());";
    }
    else{
        if(typeName.endsWith("_ITC")){
            code2=attrName+".resume();";
        }
        else{
            code2=a.Name+"=("+typeName+
            ")StateProvisioning.getData("\n"+attrName+"\n");";
        }
    }
}

resFragment2 = ${{}${}};  
  resFragment2.append(code2);  
resumeprocessor.afterEntry(resFragment2);  
}  
}  

//add the checkpoint to the main function  
//declaration of the code Fragment of result of code  
afterModified=${public static void main (String[] args){}$;  
afterModified.append("\nStateProvisioning.selectMode();\n");  
position=0;  
beforeCode=[];  
needAddedCheckpoint=[];  

//generate the new code block for the switch statement  
foreach frag in m.getBody().getFragments() do{  
    needAddedCheckpoint.add(frag);  
}  
addedToModified=  
splib.addCheckpoint(needAddedCheckpoint).getSourceString();  
afterModified.append(addedToModified+"\n");  
m.delete();  
c.addToMembers(afterModified);  
}  
}  
}  

splib.ijl:  
query library SP_Lib {  
    init {  
        console("adding the checkpoint...");  
    }  
    /**  
    *This adds the checkpoint add the checkpoint to the list of fragments  
    **/  
    public addInitilizer(declarations){  
        if(declarations.getStaticType().Name.equals("int"){  
            return "0";  
        }  
    }
else if(declarations.getStaticType().Name.equals("int")) {
    return "0";
}
else if(declarations.getStaticType().Name.equals("byte")) {
    return "0";
}
else if(declarations.getStaticType().Name.equals("short")) {
    return "0";
}
else if(declarations.getStaticType().Name.equals("long")) {
    return "0";
}
else if(declarations.getStaticType().Name.equals("float")) {
    return "0";
}
else if(declarations.getStaticType().Name.equals("double")) {
    return "0";
}
else if(declarations.getStaticType().Name.equals("char")) {
    return "0";
}
else if(declarations.getStaticType().Name.equals("boolean")) {
    return "true";
}
else {
    return "null";
}
}

public addCheckpoint( codeFragments ) {
    if(codeFragments.size<2) {
        resultCode=${} $;
        foreach frag in codeFragments do {
            resultCode.append(frag);
        }
        return resultCode;
    }
    resultCode=${switch(checkpoint.levelEntry()){}} $;
    position=0; // for calculate the number of cases
    flagCount=0; // calculate the No. of the code fragments

    // generate the new code block for the switch statement
    foreach frag in codeFragments do {
        flagCount=flagCount+1;
        frontFrag=${} $;
    }
backFrag =${}$;
backFrag.append("\ncheckpoint.moveNext();");
frontFrag.append("\ncase "+position+": {");
newCode=${}$;
newCode.append(frontFrag);
newCode.append(frag);

//Never add the cp.moveNext to the last code block
if(flagCount==codeFragments.size){
    newCode.append("\n");
}
else{
    newCode.append(backFrag);
}
position=position+1;
resultCode.append(newCode);
}
resultCode.append("\n\ncheckpoint.levelExit();");
return resultCode;
Appendix B

The Configuration Profile for Tic-Tac-Toe:

```
<application>
  <name>Tic-Tac-Toe</name>
  <directory>TTT</directory>
  <id>ttt001</id>
  <version>0.0.1</version>
  <copyright>Tic-Tac-Toe</copyright>
  <user-descriptions>
    <icon>ttt</icon>
    <description locale="DEFAULT">[ Tic-Tac-Toe]</description>
    <display-name locale="DEFAULT">[ Tic-Tac-Toe]</display-name>
  </user-descriptions>
  <vendor-info>
    <vendor-name>MSI</vendor-name>
    <vendor-url>MSI</vendor-url>
    <vendor-description>A simple Game</vendor-description>
  </vendor-info>
  <catalog-property>
    <property-name>Category</property-name>
    <property-value>Game</property-value>
  </catalog-property>
</application>
```