International Journal of Smart Home
Vol.8, No.1 (2014), pp.279-292
http://dx.doi.org/10.14257/ijsh.2014.8.1.29

A Study on the SIL Codes based Java Compiler for Supporting the Java Contents in the Smart Cross Platform

YunSik Son[1], SeMan Oh[1], JaeHyun Kim[2] and YangSun Lee[2]*

[1]Dept. of Computer Engineering, Dongguk University
26 3-Ga Phil-Dong, Jung-Gu, Seoul 100-715, KOREA
{sonbug, smoh}@dongguk.edu
[2]Dept. of of Computer Engineering, Seokyeong University
16-1 Jungneung-Dong, Sungbuk-Ku, Seoul 136-704, KOREA
*statsr@skuniv.ac.kr, *Corresponding Author: yslee@skuniv.ac.kr*

Abstract

The Smart Cross Platform is a virtual machine based solution that supports various programming languages and platforms, and its aims are to support programming languages like C++, Java and Objective-C and smart phone platforms such as Android and iOS. Various contents that developed by supported language on the Smart Cross Platform can be execute on Android and iOS platforms at no additional cost, because it has the platform independent characteristic by using SIL(Smart Intermediate Language) as an intermediate language.

In this paper, we will introduce a Java compiler for the Smart Cross Platform to support Java contents. Proposed compiler translates given Java programs into stack based intermediate SIL codes to execute on the SVM(Smart Virtual Machine). Thus, existing JVM's contents can be easily ported and executed on the Android or iOS with SVM.

Keywords: *Smart Cross Platform, Smart Virtual Machine, Smart Intermediate Language, SIL Codes based Java Compiler, Stack based SIL Codes*

1. Introduction

The previous development environments for smart phone contents are needed to generate specific target code depending on target devices or platforms, and each platform has its own developing language. Therefore, even if the same contents are to be used, it must be redeveloped depending on the target machine and a compiler for that specific machine is needed, making the contents development process very inefficient. The Smart Cross Platform is a virtual machine based solution which aims to resolve such problems, and it uses the SIL (Smart Intermediate Language) code which designed by our research team as an input at the execution time[1-4].

In this study, a compiler for use in a program designed in the Java programming language[5] to be used on the Smart Cross Platform is designed and implemented. In order to effectively implement the compiler, it was designed to five modules; syntax analysis, class file loader, symbol information collector, semantic analyzer and code generator.

This paper introduces the Java to SIL compiler in the following order. First, in Section 2, the Smart Cross Platform, SIL, and SAF are introduced. Following this, the entire composition of the compiler is introduced and the individual modules are explained in Section 3. In Section 4, we show the Java to SIL compiler's implementation and experiments. Finally, in Section 5, the results of the study and future research directions are discussed.

ISSN: 1975-4094 IJSH

2. Related Studies

2.1. Smart Cross Platform

The Smart Cross Platform developed to support downloading and executing application programs without platform dependency in the various smart devices. Another purpose of the Smart Cross Platform is multiple programming languages supporting. It's possible to support by using the intermediate language named SIL witch designed to cover both procedural programming languages and object oriented programming languages. Currently, the platform supports C/C++, Objective-C, and Java which are the most widely used languages used by developers[1-4].

The Smart Cross Platform consists of three main parts; compiler, assembler and virtual machine. It is designed in a hierarchal structure to minimize the burden of the retargeting process. Figure 1 shows a model of the Smart Cross Platform.

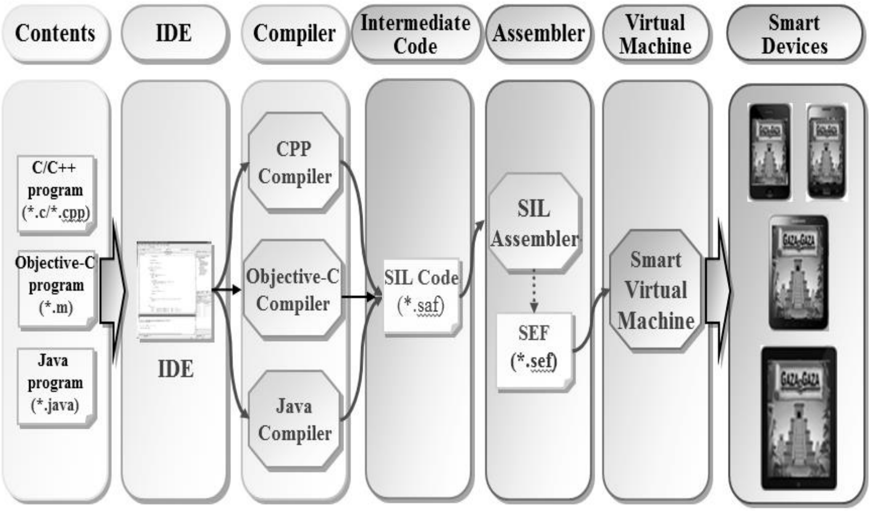


Figure 1. System Configuration of the Smart Cross Platform

The SIL code is a result of the compilation process and it is changed into the executing format SEF (SIL Executable Format) through an assembler. The Smart Virtual Machine (SVM) then runs the program after receiving the SEF. The SVM is composed by five major modules - SEF loader, stack based interpreter, SVM built-in library, native interface, runtime environments, and runtime environments consist of exception handler, memory management, and thread scheduler. And the SVM is designed to easily add debugging interface, profiling interface, and etc. The SVM’s system configuration is shown in Figure 2.

2.2. SIL (Smart Intermediate Language)

The SIL, the virtual machine code for the SVM, is designed as a standardized virtual machine code model for ordinary smart phones and embedded systems [6]. The SIL is a stack based command set which holds independence as a language, hardware and a platform.

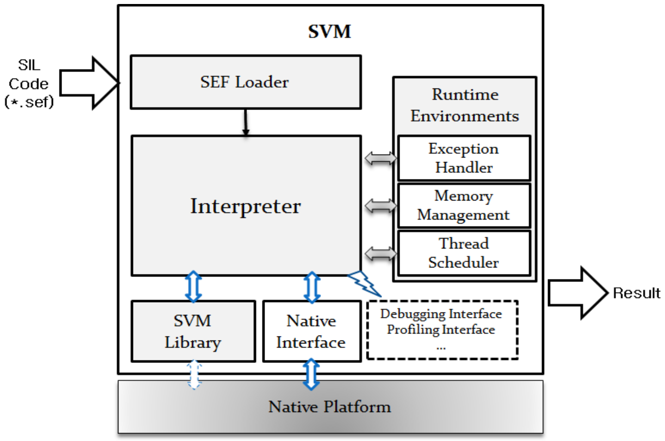


Figure 2. System Configuration of the Smart Virtual Machine

In order to accommodate a variety of programming languages, the SIL is defined based on the analysis of existing virtual machine codes such as bytecode [7-9], .NET IL [10,11] and etc. In addition, it also has the set of arithmetic operations codes to support object-oriented languages and successive languages.

The SIL is composed of meta-code (shows class declarations and specific operations) and arithmetic codes (responds to actual commands). Arithmetic codes are not subordinate to any specific hardware or source languages and thus have an abstract form. In order to make debugging of the languages such as the assembly language simple, they apply a name rule with consistency and define the language in mnemonics, for higher readability. In addition, they have short form arithmetic operations for optimization. The SIL's arithmetic codes are classified into seven categories and each one has its own detailed subcategories. Figure 3 shows categories of the SIL's operation codes.

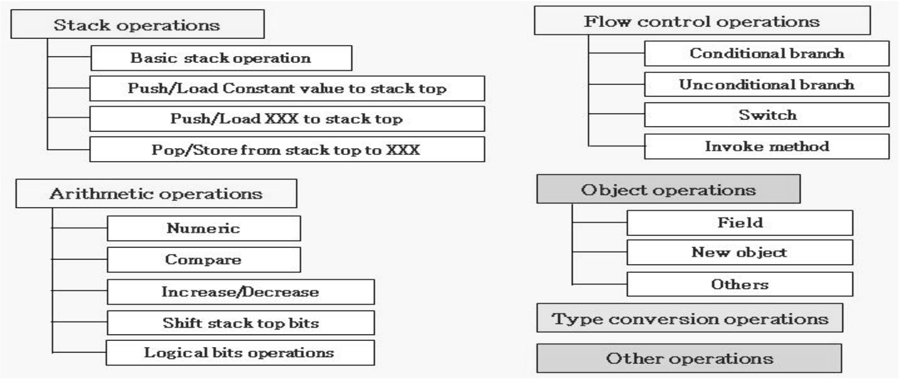


Figure 3. Category of the SIL Operation Codes

2.3. Smart Assembly Format (SAF)

The code created using high level programming language is converted into SVM's assembly format, through the code converter. The SAF format consists of pseudo code and operation code. This is then converted into a Smart Executable Format (SEF) through the assembler and is run using the SVM regardless of the system's operating system or structure. Table 1 shows the descriptions of SAF's major mnemonics.

The SAF includes a pseudo code which carries out class creation and other specific jobs and an operation code which responds to the actual commands run in the virtual machine. The operation code is a set of stack based commands which is not subordinate to specific programming languages, therefore possessing language independence, hardware independence and platform independence. As a result, an operation code's mnemonic has an abstract form as it is not subordinate to any specific hardware or source languages [12-14].

Table 1. Selected Major Mnemonics for the SAF

Mnemonic	Description
%%HeaderSection[Start/End]	Define the range of the header section.
%%CodeSection[Start/End]	Define the range of the code section.
%%DataSection[Start/End]	Define the range of the data section.
%%DebugSection[Start/End]	Define the range of the debug section.
%DefinedLiteralCount	Number of literals.
%IntializedVariableCount	Number of initialized global variables.
%UninitializedVariableCount	Number of uninitialized global variables.
%ExternalVariableCount	Number of external variables.
%ExternalFunctionCount	Number of external functions.
%InitFunctionName	Name of the initialize function for object.
%EntryFunctionName	Name of the entry point function for program execution.
%SourceFileName	Describe the program source file name.
%Function[Start/End]	Define the range of the function.
%Label	Describe the program source file name.
%Line	Describe the program source file name.
%LiteralTable[Start/End]	Define the range of the literal table section.
%InternalSymbolTable[Start/End]	Define the range of the internal symbol table section.
.func_name	Describe the function name.
.func_type	Describe the function types.
.param_count	Describe the number of parameters for the function.
.opcode_[start/end]	Define the range of the operation code section.

3. Java to SIL Compiler

In this paper, the Java to SIL compiler was designed as can be seen in Figure 4. it has five parts and 10 detailed modules.

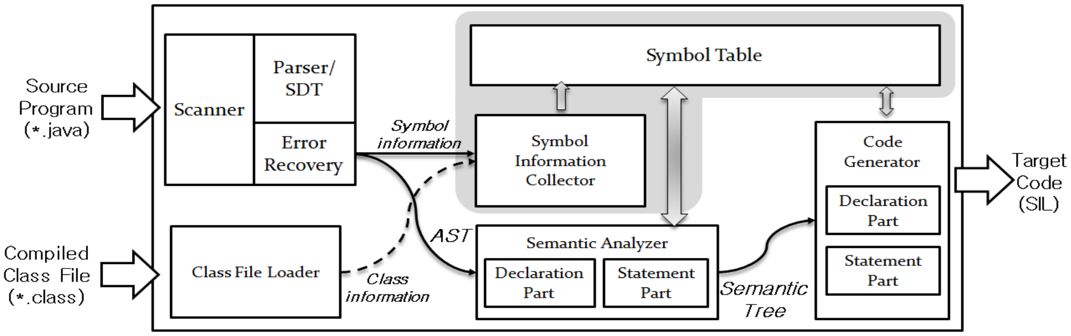


Figure 4. Java to SIL Compiler Model

The Java to SIL compiler embodies the characteristics of the Java language and therefore was designed with five different parts; syntax analysis, class file loader, symbol information collection module, semantic analysis and code generation. The detailed information for each part is as follows.

The syntax analysis part carries out syntactic analysis regarding the given input program (*.java) and converts it into an AST(Abstract Syntax Tree) which holds the equivalent semantics. There are largely three steps in the syntax analysis part; lexical analysis, syntax analysis and error recovery [15, 16]. A detailed modules relationship is shown in Figure 5.

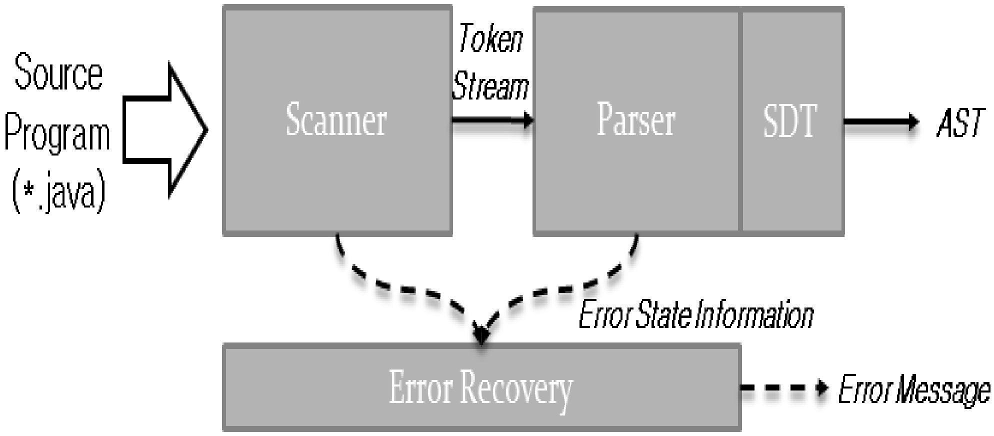


Figure 5. Syntax Analysis Module Configuration

The Class file loader is the module to extract symbol information needed to syntax analysis, semantic analysis and code generation from the pre-compiled class files. The Class file loader extracts class information from the inputted class files (*.class), and stores it in the symbol table through symbol information collector.

The module for symbol information collection consists of symbol information collection routines and a symbol table. First, the symbol information collection routine carries out the job of saving information into the symbol table which is obtained by inserting ASTs and rounding the tree. The routines consist of the interface, protocol, class member, ordinary declarations and others (given the characteristics of the Java language). Next, the symbol table is used to manage the symbols(names) and information on the symbols within a program.

The semantic analysis part is composed of the declarations semantic analysis module and the statements semantic analysis module. The declarations semantic analysis module checks the process of collecting symbol information on the AST level, to verify cases which are grammatically correct but semantically incorrect. Semantic analysis of the declarations part is handled by two parts; semantic error and semantic warning. The statements semantic analysis module uses the AST and symbol table to carry out semantic analysis of statements and creates a semantic tree as a result using the tree transformation model like Figure 6. A semantic tree is a data structure which has semantic information added to it from an AST [17-20]. It is responsible for all that has not been taken care of during the syntax analysis process and then it is used to generate codes as it has been designed to generate codes easily.

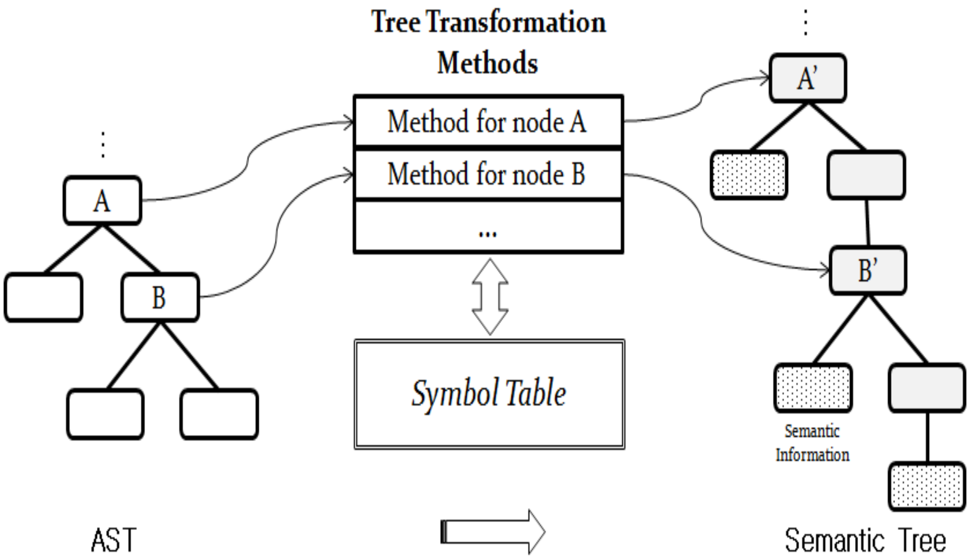


Figure 6. Tree Transformation Model

The code generation part receives the semantic tree as an input after all analysis is complete and it generates a SIL code which is semantically equal to the input pro-gram (*.java). For this, the SIL code is expressed as symbols so it is convenient to generate and handle them. For type conversion code lists, the same data structure is kept so that the code generation process can take place efficiently. Type conversion code lists are data structures that pre-calculate the process of converting a semantic code into a SIL code when generating a code. A code generator visits each nodes of the semantic tree to convert them into SIL codes.

4. Implementation and Experimental Results

To implement the Java to SIL compiler, first the language's grammar was chosen and then using this a LALR(1) parsing table was created. The grammar used was based on JDK 6.0 and the information on the grammar parsing table can be seen in Table 2. A Java grammar file segment in follow Table 3 is a Parser Generating System (PGS) input format written in LALR(1). As shown in grammar file, Java program consist of package, import, and type declarations. And type declaration the main feature defined by class and interface declarations.

Table 2. Java Grammar, Parsing Table, Tree Information

Name	Count
Grammar Rules	380
Terminal Symbols	105
Nonterminal Symbols	152
Parsing Table Kernels	650
AST Nodes	153
Semantic Tree Nodes	236

Table 3. Java Grammar File

```
SYNTAX JavaGrammer

JavaGrammer-> CompilationUnit                                   => PROGRAM;

CompilationUnit -> PackageDeclaration ImportDeclarationList TypeDeclarations;
                 -> PackageDeclaration ImportDeclarationList;
                -> PackageDeclaration TypeDeclarations;
                -> PackageDeclaration;
                -> ImportDeclarationList TypeDeclarations;
                -> ImportDeclarationList;
                -> TypeDeclarations;
                -> ;

 ImportDeclarationList -> ImportDeclarations
                                                                => IMPORT_DCL_LIST;
 ImportDeclarations -> ImportDeclaration;
                     -> ImportDeclarations ImportDeclaration;
 TypeDeclarations -> TypeDeclaration;
                    -> TypeDeclarations TypeDeclaration;
 PackageDeclaration -> 'package' Name ';'                       => PACKAGE_DCL;
 ImportDeclaration -> SingleTypeImportDeclaration;
                     -> TypeImportOnDemandDeclaration;

 SingleTypeImportDeclaration -> 'import' Name ';'               => SINGLE_IMPORT;
 TypeImportOnDemandDeclaration -> 'import' Name '.' '*' ';'     => QUALIFIED_IMPORT;
 TypeDeclaration -> ClassDeclaration;
                 -> InterfaceDeclaration;
                 -> ';';
…
```

Table 4 shows a parsing table code which generated by PGS using the Java grammar file in Table 3 as a input. The parsing table code is C source program, and it includes symbol information, length list of right hand side for shift-reduce parsing, and table for parsing action information.

Table 4. Java Parsing Table Code

```
int leftSymbol[NO_RULES+1] = {
          191,  190,  137,  137,  137,  137,  137,  137,  137,  137,
          176,  177,  177,  245,  245,  209,  175,  175,  224,  246,
         ...
          164,  219,  219,  133,  134,  187,  115,  115,  115,  206,
          206,  206,  223,  218,  218,  136};

 int rightLength[NO_RULES+1] = {
            2,    1,    3,    2,    2,    1,    2,    1,    1,    0,
            1,    1,    2,    1,    2,    3,    1,    1,    3,    5,
         ...
            1,    1,    1,    1,    1,    1,    3,    3,    3,    1,
            1,    1,    1,    3,    3,    3};
```

```
int parsingTable[NO_STATES][NO_SYMBOLS+1] = {
        {/*** state    0 ***/
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,  30,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,  -9,  29,   0,   0,   0,   0,
        ...
          0,   0,   0,   0,   3,   2,   1,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0},
        {/*** state    1 ***/
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,
        ...
```

Table 5. Example Program(TicTacToe.java)

```
public class TicTacToe extends Component
{
    ...
  public TicTacToe() {
    this.player = 0;
    this.computer = 0;
    ...
  }

  public void paint( Graphics g) {
    g.setColor( getBackground());
    g.fillRect( 0, 0, getSize().width,
                getSize().height);
    g.setColor( gridColor);
    int fieldSize = this.getFieldSize();
    g.drawLine( 0, fieldSize, 3*fieldSize,
                fieldSize);
    g.drawLine( 0, fieldSize+1,
                3*fieldSize, fieldSize+1);
    ...
  }
  ...
}

…
```

Next, we show the process of converting the source program's code(written in Java language) into the target code, the SIL code, using the implemented Java to SIL compiler. Table 5 has been created so that the characteristics of the declarations and syntax of the example program can be seen using the Java language.

Table 6 shows the AST structures generated from the input program. You can see that the syntax have been expressed using the AST nodes defined earlier on. Table 7 shows a part of the SIL code that has been generated using a semantic tree.

Table 6. AST for an Example Program Segment

```
Nonterminal: PROGRAM
…
  Nonterminal: CLASS_DCL
    Nonterminal: PUBLIC
      Terminal: TicTacToe
    Nonterminal: EXTENDS
      Nonterminal: CLASS_
            INTERFACE_TYPE
        Nonterminal:
SIMPLE_NAME
  Terminal: Component
Nonterminal: CLASS_BODY
  Nonterminal: FIELD_DCL
    Nonterminal: PRIVATE
    Nonterminal: DCL_SPEC
      Nonterminal: INT_TYPE
    Nonterminal: VAR_ITEM
      Nonterminal:
             SIMPLE_VAR
      Terminal: player
  Nonterminal: FIELD_DCL
    Nonterminal: PRIVATE
    Nonterminal: DCL_SPEC
      Nonterminal: INT_TYPE
    Nonterminal: VAR_ITEM
      Nonterminal:
            SIMPLE_VAR
     Terminal: computer
    …
```

Table 7. Generated SIL Code for Example Program

```
%%HeaderSectionStart
…
%%HeaderSectionEnd
%%CodeSectionStart
  %FunctionStart
    .func_name
          &TicTacToe::TicTacToe$0
    .func_type   2
    .param_count   0
    .opcode_start
      proc   16   1   1
      lod.p   1   0
      ldc.p
    add.p
    ldc.i   0
    sti.i
    lod.p   1   0
    ldc.p   4
    add.p
    ldc.i   0
    sti.i
%Label   ##0
    lod.i   1   4
    lod.i   1   0
    le.i
    fjp   ##1
    ldc.i   0
    str.i   1 12
    ldc.i   1
    str.i   1 8
%Label   ##3
    lod.i   1   8
    lod.i   1   4
    ldc.i   2
    div.i
    le.i
    fjp   ##4
    lod.i   1   4
    lod.i   1   8
    mod.i
    ldc.i   0
    eq.i
    fjp   ##6
    lod.i   1   12
    str.i   1   12
    add.p
    ldi.p
    …
    .opcode_end
  %FunctionEnd

…
%%CodeSectionEnd
%%DataSectionStart
…
%%DataSectionEnd
    lod.i   1   8
    add.i
```

Next Table 8 shows the experimental result of implemented compiler. We select the example files to test main features of the Java programming language like class, sub class, method overloading, and interface/abstract class.

Table 8. AST for an Example Program Segment

(a) Class Example

Source code

```
class Fraction {
   int numerator, denominator;
   Fraction(int num, int denom) {
      numerator = num;
      denominator = denom; }
   public void printFraction() {
      System.out.println(numerator
          + "/" + denominator); }
}
public class SubPartOfFraction {
   public static void main(String[] args) {
      Fraction f = new Fraction(1, 2);
      f.printFraction(); }
}
```

SIL Code / Execution Result

```
%%HeaderSectionStart
 ... omitted ...
%SourceFileNameSubPartOfFraction.java
%%HeaderSectionEnd
%%CodeSectionStart
 ... omitted ...
%FunctionStart
.func_name&Fraction.printFraction
.func_type2
.param_count0
.opcode_start
procva411
str.p10
ldp
ldp
ldp
lod.i10
lda0@0
calls8
lod.i14
calls8
... omitted ...
```

```
C:₩test>svm SubPartOfFraction
1/2
```

(b) Sub Class Example

Source code

```
class SuperClass { int a = 1; int b = 1; }
class SubClass extends SuperClass {
   int a = 2;  double b = 2.0;
   void output() {
      System.out.println("Base class: a = "
          + super.a + ", Extended class: a = " + a);
      System.out.println("Base class: b = "
          + super.b +", Extended class: b = " + b); }
}
public class NameConflict {
   public static void main(String[] args) {
      SubClass obj = new SubClass();
      obj.output();
   }
}
```

SIL Code / Execution Result

```
 ... omitted ...
%FunctionStart
.func_name&SubClass.output
.func_type2
.param_count0
.opcode_start
procva411
str.p10
ldp
ldp
ldp
ldp
lda0@0
ldfld.i10
calls8
lda0@4
calls8
ldfld.i18
calls8
calls217
... omitted ...
```

```
C:₩test>svm NameConflict
Base class: a = 1, Extended class: a = 2
Base class: b = 1, Extended class: b = 2.0
```

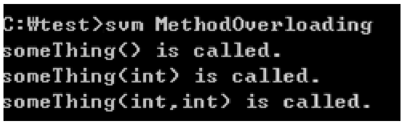
(c) Method Overloading Example

Source code	SIL Code / Execution Result

```
public class MethodOverloading {
   void someThing() {
      System.out.println("someThing() is called.");
   }
   void someThing(int i) {
     System.out.println(
        "someThing(int) is called.");
   }
   void someThing(int i, int j) {
      System.out.println(
         "someThing(int,int) is called.");
   }
   public static void main(String[] args) {
      MethodOverloading m
          = new MethodOverloading();
      m.someThing();
      m.someThing(526);
      m.someThing(54, 526);
   }
}
```

```
 ... omitted ...
%FunctionStart
.func_name&Method
   Overloading.
   someThing$1
.func_type2
.param_count1
.opcode_start
procva411
str.p10
ldp
lda0@1
```

```
calls217
 ... omitted ...
ret
.opcode_end
%FunctionStart
.func_name&Method
   Overloading.
   someThing$2
.func_type2
.param_count2
.opcode_start
... omitted ...
```



```
C:₩test>svm MethodOverloading
someThing() is called.
someThing(int) is called.
someThing(int,int) is called.
```

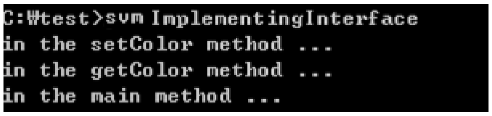
(d) Interface, Abstract Class Example

Source code

```
interface BaseColors {
   int RED = 1, GREEN = 2, BLUE = 4;
   void setColor(int color);
    int getColor();
 }
abstract class SetColor implements BaseColors {
   protected int color;
   public void setColor(int color) {
      this.color = color;
       System.out.println("in the setColor method ...");
   }
}
class Color extends SetColor {
   public int getColor() {
      System.out.println("in the getColor method ...");
      return color;
   }
}
public class ImplementingInterface {
   public static void main(String[] args) {
      Color c = new Color();
      c.setColor(10);
      int i = c.getColor();
      System.out.println("in the main method ...");
   }
}
```

SIL Code / Execution Result

```
                  ... omitted ...
%FunctionStart
.func_name&ImplementingInterface.main
.func_type2
.param_count1
.opcode_start
procva414
str.p10
callColor.Color$0
sta10
ldp
lda10
ldc12
add.p
calli
str.i14
                  ... omitted ...
```



```
C:₩test>svm ImplementingInterface
in the setColor method ...
in the getColor method ...
in the main method ...
```

Table 9. Execution Result of the Sample Game Content on the Smart Cross Platform(Android Ver.)

Source code	Execution Result

```
class FingerRunner {
   class ContentsThread extends Thread {
      void EVENT_START ()
      {
         ContentsGlobalVar.f_cx
             = GnexGlobalVar.swWidth >> 1;
         ContentsGlobalVar.f_cy
             = GnexGlobalVar.swHeight >> 1;
         if (GnexGlobalVar.swHeight > 800)
            ContentsGlobalVar.PosTop
                = ContentsGlobalVar.f_cy - 400;
         else
            ContentsGlobalVar.PosTop = 0;
         ReadRom();
         ContentsGlobalVar.f_state
             = ContentsGlobalVar.FR_LOGO;
         InitLogo();
         InitTouchArea();
         GnexApi.SetTimer(2, 1);
         GnexApi.SetTimer1(500, 0);
      }
   …
```

Table 9 shows the execution result for the Java game content. The experimental SVM is ported on the Android platform, and the content is executed on the SVM in the Android.

5. Conclusions

Virtual machines refer to the technique of using the same application program even if the process or operating system is changed. It is the core technique that can be loaded onto recently booming smart phones, necessary as an independent download solution software technique.

In this study, the Java to SIL compiler was designed and implemented to run the target contents that was originally created for another platform to enable its use on the Smart Cross Platform. In this paper, we defined five modules to create a compiler and generate a SIL code for use on the SVM which is independent of platforms. As a result, programs developed for use as Java contents could be run on the SVM using the compiler developed throughout the study and therefore expenses required when producing such contents can be minimized.

In the future, there is need for research on an Android Java to SIL compiler so that Android contents can be run by the Smart Cross Platform. Further research on optimizers and assemblers for SIL code programs are also needed so that SIL codes that have been generated can execute effectively on the SVM.

Acknowledgements

This paper was extended from the previous research paper “Design and Implementation of the Java Compiler for the Smart Virtual Machine Platform” in MulGraB 2012.

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology(No. 20110006884), and in part by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT and future Planning(No. 2013R1A2A2A01067205).

References

[1] Y. S. Son and Y. S. Lee, “A Study on the Smart Virtual Machine for Smart Devices”, Information-an International Interdisciplinary Journal, International Information Institute, vol. 16, no. 1465, **(2013)**.
[2] S. M. Han, Y. S. Son and Y. S. Lee, “Design and Implementation of the Smart Virtual Machine for Smart Cross Platforms”, Journal of Korea Multimedia Society, Korea Multimedia Society, vol. 16, no. 190, **(2013)**.
[3] Y. S. Lee and Y. S. Son, “A Study on the Smart Virtual Machine for Executing Virtual Machine Codes on Smart Platforms”, International Journal of Smart Home, SERSC, vol. 6, no. 93, **(2012)**.
[4] Y. S. Lee, “The Virtual Machine Technology for Embedded Systems”, Journal of Korea Multimedia Society, Korea Multimedia Society, vol. 6, no. 36, **(2002)**.
[5] The Java Language & Virtual Machine Specifications, Oracle, http://docs.oracle.com/javase/specs/index.html.
[6] S. L. Yun, D. G Nam, S. M. Oh and J. S Kim, “Virtual Machine Code for Embedded Systems”, International Conference on CIMCA, vol. 206, **(2004)**.
[7] J. Meyer and T. Downing, “JAVA Virtual Machine”, O'REYLLY, **(1997)**.
[8] T. Lindholm and F. Yellin, “The Java^{TM} Virtual Machine Specification”, 2nd ed., Addison Wesley, **(1999)**.
[9] J. Engel, “Programming for the Java Virtual Machine”, Addison-Wesley, **(1999)**.
[10] S. Lindin, “Inside Microsoft .NET IL Assembler”, Microsoft Press, **(2002)**.
[11] Microsoft, MSIL Instruction Set Specification, Microsoft Corporation, **(2000)**.
[12] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, “Compilers: Principles, Techniques, & Tools. Addison-Wesley”, **(2007)**.
[13] D. Grune, H. E. Bal, C. J. H. Jacobs and K. G. Langendoen, “Modern Compiler Design”, John Wiley & Sons, **(2000)**.
[14] S. M. Oh, “Introduction to Compilers”, 3rd edition, Jungik Publishing, Seoul, Republic of Korea, **(2006)**.
[15] I. S. Kim and K. M. Choe, “Error Repair with Validation in LR-Based Parsing”, ACM Transactions on Programming Languages and Systems, ACM, vol. 23, no. 451, **(2001)**.
[16] S. L. Graham, C. B. Haley and W. N. Joy, “Practical LR Error Recovery”, Proceedings of the SIGPLAN Symposium on Compiler Construction, ACM, vol. 13, no. 168, **(1979)**.
[17] J. M. Barth, “A Practical Interprocedural Data Flow Analysis Algorithm”, Communications of the ACM, 21, ACM, 724, **(1978)**.
[18] Y. S. Son, “2-Level Code Generation using Semantic Tree”, Master Thesis, Dongguk University, **(2006)**.
[19] Y. S. Son and Y. S. Lee, “An Objective-C Compiler to Generate Platform-Independent Codes in Smart Device Environments”, Information: An International Interdisciplinary Journal, International Information Institute, vol. 16, no. 1459, **(2012)**.
[20] Y. S. Son and Y. S. Lee, “A Study on the Java Compiler for the Smart Virtual Machine Platform”, Communications in Computer and Information Science (CCIS), Springer, vol. 353, no. 135, **(2012)**.

Authors

YunSik Son, he received the B.S. degree from the Dept. of Computer Science, Dongguk University, Seoul, Korea, in 2004, and M.S. and Ph.D. degrees from the Dept. of Computer Engineering, Dongguk University, Seoul, Korea in 2006 and 2009, respectively. Currently, he is a Researcher of the Dept. of Computer Science and Engineering, Dongguk University, Seoul, Korea. His research areas include smart system solutions, secure software, programming languages, compiler construction, and mobile/embedded systems.

SeMan Oh, he received the B.S. degree from the Seoul National University, Seoul, Korea, in 1977, and M.S. and Ph.D. degrees from the Dept. of Computer Science, Korea Advanced Institute of Science and Technology, Seoul, Korea in 1979 and 1985, respectively. He was a Dean of the Dept. of Computer Science and Engineering, Graduate School, Dongguk University from 1993-1999, a Director of SIGPL in Korea Institute of Information Scientists and Engineers from 2001-2003, a Director of SIGGAME in Korea Information Processing Society from 2004-2005. Currently, he is a Professor of the Dept. of Computer Science and Engineering, Dongguk University, Seoul, Korea. His research areas include smart system solutions, programming languages, and embedded systems.

JaeHyun Kim, he received the B.S. degree from the Dept. of Mathematics, Hanyang University, Seoul, Korea, in 1986, and M.S. and Ph.D. degrees from Dept. of Statistics, Dongguk University, Seoul, Korea in 1989 and 1996, respectively. He was a chairman of Dept. of Internet Information 2002-2007. Currently, he is a member of the Korean Data & Information Science Society and a Professor of Dept. of Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include mobile programming, cloud system and data analysis.

YangSun Lee, he received the B.S. degree from the Dept. of Computer Science, Dongguk University, Seoul, Korea, in 1985, and M.S. and Ph.D. degrees from Dept. of Computer Engineering, Dongguk University, Seoul, Korea in 1987 and 2003, respectively. He was a Manager of the Computer Center, Seokyeong University from 1996-2000, a Director of Korea Multimedia Society from 2004-2014, a General Director of Korea Multimedia Society from 2005-2006 and a Vice President of Korea Multimedia Society in 2009. Also, he was a Director of Korea Information Processing Society from 2006-2013 and a President of a Society for the Study of Game at Korea Information Processing Society from 2006-2010. And, he was a Director of Smart Developer Association from 2011-2014. Currently, he is a Professor of Dept. of Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include smart system solutions, programming languages, and embedded systems.