International Conference
FROM DELC TO VELSPACE
Plovdiv, 26–28 March 2014

SOLIDREFLECTOR: A MULTISTAGE, INTERACTIVE DECOMPILATION FRAMEWORK

Vassil Vassilev, Martin Vassilev, Petya Petrova

Abstract. *The paper describes a multistage, interactive analysis and decompilation framework – SolidReflector. Some of the key algorithms responsible for the generation of intermediate representations are outlined and accompanied with examples. The work presents loose-coupled visualization techniques pairing code representations with their visual counterparts. We identify the benefits and the applications of the multistage decompilation. We discuss the pros of the interactivity of the decompiler in areas such as university education.*

Keywords: decompiler, multistage decompilation
Mathematics Subject Classification 2010: 68N20, 97R99

1. INTRODUCTION

Good understanding and analysis of the software are essential for computer science. The tendency of increasing codebases and the complexity of the software systems outlines already the profile of the contemporary and future computer specialists. A common denominator usually is the ability to understand quickly complex systems that may be implemented and integrated decades ago. Most of the time tools such as text editors, static analyzers and IDEs can help. In university these tools are used to show the students various aspects of the practical materials. Very often the tools are static and able to show only one layer of information, such as the source code of an application.

There are multiple layers between the source and machine code. Considerable part of the information flow in the process of translation remains hidden. Inevitably it leads to the troubled understanding, which approach is the most efficient in order to implement the desired behavior. This becomes a central issue with the modern compilers, which use multi-stage compilation [1]. Understanding the different stages is not easy not only for students but for experts, too.

A goal of the developed framework is to reveal the process of translation of the high-level source code to machine executable code by inverting the compilation chain. Thus, if we simplified, we could classify the tool as a decompiler and the framework as a decompilation framework. In this paper we will try to prove this

would be an oversimplification. Alongside with the framework we provide a standalone tool, demonstrating features of the framework.

The paper is divided as follows: Section 2, related work; Section 3, architecture and concepts, discussing the challenges in building a multistage, open and user-friendly system; Section 4, implementation, motivates the concrete realization details and taken decisions; Section 5, application scenarios, suggests possible applications of the work in a few domains; and Section 6, conclusion, briefly summarizes the presented work and gives future perspectives of the project.

2. RELATED WORK

Classification of the framework is not easy. If we said it is a decompilation toolchain, we would need to review a few decompilers. SolidReflector builds flow graphs, which are very common for the static analyzers.

.NET Reflector is a class browser, decompiler and static analyzer for .NET based applications [2]. It is the first assembly browser based on CLI (Common Language Infrastructure) and it is capable of working with every .NET/Mono-targeted assemblies. The decompiler recreates readable high-level source code, but it does not reproduce suitable and informative code models. Another disadvantage is the fact that .NET Reflector is a commercial product and it is not platform independent. Moreover, it lacks interactive manipulation of the loaded assemblies.

Since .NET Reflector became commercial product the IL Spy project was started. This is an open source tool used to browse and decompile C# assemblies including ones with C# version 4.0 and 5.0 [3]. Disadvantage of IL Spy is the lack of interactivity, the absence of multiple code model generation and the platform dependence on Windows.

Most of the available code analyzers focus on just analyzing the source code, their common feature is restricted to providing the read only abilities of previewing and browsing the code and information related to the loaded executables. They provide only static and non-interactive analyses. Very few show the multistage translation layers. Moreover, these tools are oriented towards the expert, turning them into not very appropriate for education purposes.

3. ARCHITECTURE & CONCEPTS

The system has many diverse ingredients coming from different domains, such as compiler construction, simulation and data visualization. Loose coupling becomes a must in order to separate concerns and provide a good extensible model. The implementation follows the MVC pattern [4] and provides a modular subsystem.

Many aspects of the translation process require code models (intermediate representations) at different levels of abstraction. The problem is that the above-

mentioned types of representations are challenging to understand when displayed in raw form. There are not enough tools, which can show multiple layers of information about certain piece of software, and allowing the user to interactively browse and change the displayed contents. The implementation of interactive application analyzers would be greatly beneficial in the field of education as well.

SolidReflector is a tool developed in the context of SolidOpt [5]. SolidOpt is a framework for carrying out automatic optimizations, developed by the same research group. It is capable of analysing executable code and based on it to create multiple models of the code at different levels of abstraction. SolidReflector uses SolidOpt as a library, in order to build the needed representations of a .NET assembly. Once the representation is built, it is extended with a graphical visualization, showing implicit information such as nodes and edges in the flow graphs, for example. Visual and non-visual code models are bound together to create a hybrid graph of representations [5]. It allows the built representations to be changed in flight. The changed representations can be lowered to an executable and their execution can be simulated in a secure environment. A major advantage of this setup is that it makes the multistage compilation a little more comprehensive.

Understanding how the representation of the program works usually needs trial and error learning. This implies our representations to be mutable and to execute the changes. In combination with user-friendly organisation of the interface the representation turns into an interactive representation. Many of the built representations in SolidReflector have interactive layer making them easy to change and comprehend.

SolidReflector contains various graphical primitives ready to be used for as building blocks to display the representations' data. The primitives can be inherited, modified and grouped into new composite forms in order to achieve better exposition of data.

After a change in a representation by the user, the modifications have to be propagated down to the assembly, i.e. to the executable code. In order to verify effect of the modification, the user could test it within an isolated scope by performing a simulative execution. Due to security reasons the assembly execution is rather simulated than executed, i.e. the application is never executed natively, but in a controlled environment. It is implemented by creating a dedicated application domain with restricted permissions, where the execution takes place.

For instance, changing the nodes of method's control flow graph can alter the semantics not only of the method itself, but the entire program. The modification of the semantics can be done only using the interactive interface and moving the links between the nodes only through the graphical user interface.

SolidReflector uses a plugin-based architecture and it can be divided into a core and plugins [6], [7]. The core can be described as the mechanism responsible for providing base infrastructure suitable for manipulating and loading plugins. It provides a basic graphical interface that can be used for docking different types of controls.

4. IMPLEMENTATION

The tool is written in C# and can run on multiple platforms such as Linux, Mac and Windows. It can provide descriptive information about CLI [8] assemblies by building representations of the code such as control flow graphs, call graphs and three address code.

4.1. ASSEMBLY BROWSER PLUGIN

The Assembly Browser stores multiple loaded assemblies (❶ on Figure 1). It provides a convenient tree-like hierarchy representation of the assembly data. The tree consists of four levels. The first level shows the assembly name; the second level shows the list of modules defined by the assembly; the third level shows the list of types defined by the module; the fourth level – the list of methods, fields and events defined by the type. There is a changes monitoring mechanism implemented that is observing each assembly. If a loaded assembly is externally modified a warning in the application is generated and then the assembly is reloaded.

4.2. COMMON INTERMEDIATE LANGUAGE VISUALIZER PLUGIN

The loaded assemblies (❷ on Figure 1) contain various meta information regarding the assembly itself or modules, types, methods and common intermediate language (CIL) instructions used in it.

4.3. CONTROL FLOW GRAPH VISUALIZER PLUGIN

A control flow graph (CFG) is a graph representing the execution flow. Each graph node contains instructions grouped in basic blocks. Each basic block is filled with linear instructions, i.e. instructions that do not change the control flow and that are executed in a row – one after another. There is a branch or return instruction at the end of each basic block and the next instruction starts a new basic block. The edges of the built graph model all possible branches between the basic blocks.

The control flow graph (❸ on Figure 1) is a graph based intermediate representation of the CIL code. There are two types of branches:

- Structural – i.e. branches caused by the ‘normal’ possible changes in the control flow of the program;
- Exceptional – i.e. branches caused by the exceptional possible changes in the control flow of the program.

The listed pseudocode in Listing 1 gives the concept of creating and connecting the basic blocks.

```
void function CreateBlocks()
  Foreach (instr in instructions)
  {
    If IsBlockLeader(inst),
      set block to CreateNewBlock();
    block.add(instr)
    If IsBlockTerminator(instr),
      BlockList.add(block);
  }

void function ConnectBlocks()
  Foreach (block in BlockList)
  {
    Set targets to GetTargetInstructions(LastInstr)
    Foreach (target in targets)
    {
      Set succ to GetNodeContaining(target);
      block.Successors.add(succ);
      succ.Predecessors.add(block);
    }
  }
```

Listing 1: Pseudocode creating structure CFG

Table 1 presents an example of an exception-free (structure) CFG. It represents the control flow of an if-else statement (left). In the middle is shown the corresponding CIL code generated after compilation of the high-level code and on the right is the interactive representation of the control flow graph.

High-Level Code (C#)	CIL	CFG

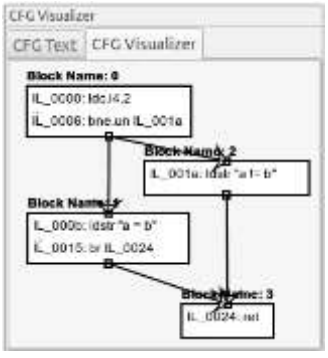
`static void Main()`
`{`
`int a = 2;`
`int b = 0;`
 	
`if (a == b)`
`  Console.Write ("a=b");`
`else`
 	
`Console.Write("a!=b");`
`}`
	.method public hidebysig static void ***Main*** *() cil managed {*
	IL_00: ***ldc.i4.2***
	IL_01: ***stloc.0***
	IL_02: ***ldc.i4.0***
	IL_03: ***stloc.1***
	IL_04: ***ldloc.0***
	IL_05: ***ldloc.1***
	IL_06: ***bne.un IL_001a***
	IL_0b: ***ldstr "a = b"***
	IL_10: ***call Write (String)***
	IL_15: ***br IL_0024***
	IL_1a: ***ldstr "a != b"***
	IL_1f: ***call Write(String)***
	IL_24: ***ret***
	}
		

Table 1: C# to structure CFG transformation

Table 2 illustrates the exception-based CFG. The represents the control flow graph of a try-catch-finally statement. Main difference in building exception CFG

is the exception handling model relies on the specifics of the design of the executor. In the case of CLR, many of the control flow rules are not part of the instruction object model. They are annotated by special instructions, which can be only used when exception is being handled. CLR treatment of those instructions sometimes is very complex and obscure, which makes part of the implementation very cumbersome and tricky.

High-Level Code (C#)	CIL	CFG

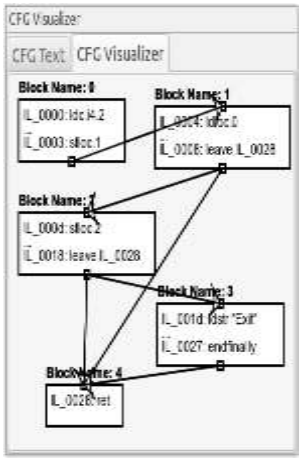
`static void Main() {`
`int a = 2;`
`int b = 0;`
 	
`try`
`{`
`a = a / b;`
`}`
 	
`catch (Exception ex)`
`{`
`Console.Write("Div by 0");`
`}`
 	
`finally`
`Console.Write("Exit");`
`}`
	.method public hidebysig static void **Main** *() cil managed {*
	IL_00: **ldc.i4.2**
	IL_01: **stloc.0**
	IL_02: **ldc.i4.0**
	IL_03: **stloc.1**
	IL_04: **ldloc.0**
	IL_05: **ldloc.1**
	IL_06: **div**
	IL_07: **stloc.0**
	IL_08: **leave IL_0028**
	IL_0d: **stloc.2**
	IL_0e: **ldstr "Div by 0"**
	IL_13: **call write(string)**
	IL_18: **leave IL_0028**
	IL_1d: **ldstr "Exit"**
	IL_22: **call write(string)**
	IL_27: **endfinally**
	IL_28: **ret**
	.try *L_0004 to L_000d* **catch** *Exception* **handler** *L_000d to L_001d*
	.try *L_0004 to L_001d* **finally** *handler L_001d to L_0028*
	}
		

Table 2: C# to exception CFG transformation

4.4. CALL GRAPH VISUALIZER PLUGIN

The call graph (❹ on Figure 1) is a representation responsible for modeling the method calls in an application [1]. The call graph (CG) contains nodes and edges, where:

- A method is represented by a node;
- Method call is represented as a node;
- An edge is created between method A and method B if A calls B.

In the right-most column in Table 3 is illustrated a call graph built for the method ‘Main’ (shown in the left-most column).

The pseudocode for the recursive function responsible for the call graph generation can be seen in the middle column of Table 3.

High-Level Code (C#)	Pseudocode	Call Graph

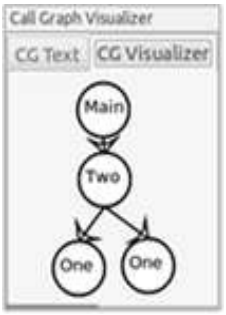
`public int Zero()`
`{`
`  return 0;`
`}`
`public int One()`
`{`
`  return 1;`
`}`
`public int Two()`
`{`
`  return One() + One();`
`}`
`public int Main()`
`{`
`  return Two();`
`}`
	`void function VisitMethod(CGNode node)`
	`ForEach (instruction in instructions)`
	`  if (instruction.opcode == MethodCall)`
	`  {`
	`  Set callee to new CGNode();`
	`  node.MethodCalls.add(callee);`
	`  VisitMethod(callee);`
	`  }`
		

Table 3: C# to Call Graph transformation

4.5. THREE ADDRESS CODE VISUALIZER

Three-address code (TAC) is an intermediate code representation where each statement contains at most one operator on the right side of an instruction (❺ on Figure 1). The three address instructions are based on two concepts – addresses and instructions. The addresses can be names; constants; or temporaries. The instructions can be: assignment instructions; copy instructions; unconditional jumps; conditional jumps; procedure calls; return instructions; array manipulation instructions; address and pointer instructions; type casts; etc. [1].

Implementing CIL to TAC transformation is not a trivial task. It needs to transform the stack-based CIL into a close to a register-based representation. Thus, the implementation requires the use of a simulation stack. The stack simulates execution of the CLR instructions by iterating over them. In brief, when the transformer encounters an instruction, whose semantics is storing information onto the stack – it pushes this information onto the simulation stack. On encountering an instruction, whose semantics is loading from the stack instruction, it takes the information from the top of the simulation stack and does the translation depending on its semantics.

On Listing 2 is illustrated how stloc.0 (store local variable on the stack) and ldloc.0 (load local variable from the stack) are decompiled.

```
Set instr to GetFirstInstruction();
while (instr not null)
{
  switch (instr.OpCode)
  {
    case Code.Stloc_0:
    triplets.Add(Triplet(TripletOpCode.Assignment,
                                GetFirstVar(), stack.Pop()));
    break;
    case Code.Ldloc_0:
      stack.Push(method.Body.Variables[0]);
    break;
  ...
  }
  set instr to instr.next()
}
```

Listing 2: Pseudocode for CIL to TAC transformation

In the example below (Table 4) is shown a CIL to TAC transformation. The three-address code representation is designed to work with the CFG builder and a CFG for the TAC could be build using the described algorithm in 4.3.

High-Level Code (C#)	CIL	TAC
`public static int Main() { int i = 0; int j; j = i++; return j; }`	`.method public hidebysig static void Main () cil managed { IL_00: ldc.i4.0 IL_01: stloc.0 IL_02: ldloc.0 IL_03: dup IL_04: ldc.i4.1 IL_05: add IL_06: stloc.0 IL_07: stloc.1 IL_08: ldloc.1 IL_09: ret }`	TAC Visualizer TAC Text System.Int32 CallGraphEx L0: V_0 = 0 L1: T_0 = V_0 L2: T_1 = T_0 + 1 L3: V_0 = T_1 L4: V_1 = T_0 L5: return V_1 }

Table 4: CIL to TAC transformation

Figure 1 shows the described tool in practice. It can build simultaneously various representations in reverse to the multistage compilation order – providing a multistage decompilation. The multistage decompilation shows very precise information about the process of translation from a high-level language to CIL and outlines the information flow in lowering high-level constructs into their low-level counterparts. This greatly improves the comprehension of the entire process.

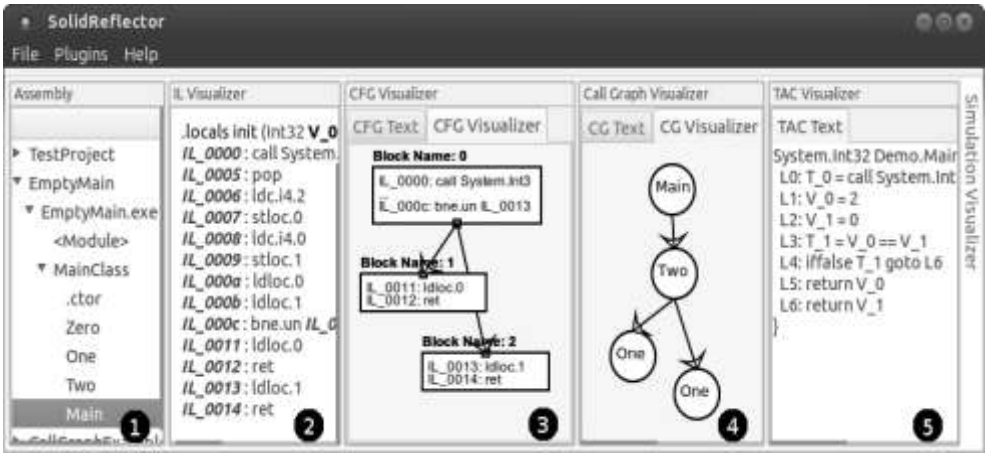


Figure 1: SolidReflector

5. CONCLUSION

SolidReflector evolves constantly and its prototype converges into a standalone, usable tool. It combines all ingredients, necessary for building an interactive tool, able to show many layers of a software system and outlining the use of reverse engineering in the education. It works on any CLR [8] assemblies and it can display different representations of the program logic alongside with interactive

visualization. It was already used to study the specifics of CLR and the quality of the generated by the compiler code. This makes it an excellent candidate as an education tool in courses such as compiler construction and performance optimizations. Another usage scenario is for experienced programmers. They could use the tool to study which high-level constructs get compiled more efficiently.

A possible future direction would be to cover other virtual machines such as the java virtual machine (JVM) and LLVM. This is not a trivial endeavor, which can broaden even more the application scenarios. There is a lot of space for future improvements but the most important ones are the ability to build an interactive abstract syntax tree (AST) representation and actual source code. The AST would greatly improve the code retargeting features, i.e. decompilation from one executable format and then translating it into another.

Improvements in the interactivity are always a vital component. Depending on information about the ways of use, the interface and commands can be further tweaked to match the most common usage setups.

REFERENCES

[1] A Aho et al., *Compilers – Principles, Techniques and Tools 2nd edition*

[2] .NET Reflector. http://www.reflector.net/ *(visited in March 2014)*

[3] Il Spy. http://ilspy.net/ *(visited in March 2014)*

[4] Krasner, G. and S. Pope, A cookbook for using the model-view controller user interface paradigm in smalltalk-80, *J. Object Oriented Program*, 1 (3), 1988, 26–49.

[5] Penev, A., Computer Graphics and Geometric Modelling – A Hybrid Approach, *Journal of Pure and Applied Mathematics*, 85 (4), 2013, 781–811.

[6] Vassilev, V. et al, SolidOpt – Innovative Multiple Model Software Optimization Framework, *IEEE and STRL: The Second Conference on Creativity and Innovations in Software Engineering*, 2009.

[7] Penev, A., D. Dimov and D. Kralchev, Open hybrid system for geometrical modeling, *In Proceedings of the 17th International conference SAER-2003 Conference*, (1), 2003, 131–135.

[8] ECMA-335, *Common Language Infrastructure (CLI)*, ISO/IEC 23271, 2012.

Faculty of Mathematics and Informatics,
University of Plovdiv "Paisii Hilendarski"
236 Bulgaria blvd, Plovdiv, Bulgaria
vvasilev@cern.ch, mrtn.vassilev@gmail.com, petya.petrova@hotmail.com

SOLIDREFLECTOR – МНОГОСТЪПКОВ, ИНТЕРАКТИВЕН ИНТРУМЕТАРИУМ ЗА ДЕКОМПИЛАЦИЯ

Васил Василев, Мартин Василев, Петя Петрова

Резюме. *В тази статия ние описваме многостъпков, интерактивно-анализиращ и декомпилилиращ инструментариум – SolidReflector. Представени са някои от основните алгоритми, отговарящи за генерирането на междинни представяния, съпроводени от примери. Демонстрирано е използването на слабо свързана система за визуализация, която свързва моделите на кода с техните съответни визуализатори. Показани са предимствата на многостъпковата декомпилация. Обсъждат се предимствата на интерактивността на декомпилатора във сфери като обучението.*