Cross-language Program Slicing in the .NET Framework

Krisztián Pócza
Eötvös Loránd University
Fac. of Informatics, Dept. of Programming Languages and Compilers
Pázmány Péter sétány 1/c.
H-1117, Budapest, Hungary
kpocza@kpocza.net

Mihály Biczó
Eötvös Loránd University
Fac. of Informatics, Dept. of Programming Languages and Compilers
Pázmány Péter sétány 1/c.
H-1117, Budapest, Hungary
mihaly.biczo@axelero.hu

Zoltán Porkoláb
Eötvös Loránd University
Fac. of Informatics, Dept. of Programming Languages and Compilers
Pázmány Péter sétány 1/c.
H-1117, Budapest, Hungary
gsd@elte.hu

ABSTRACT

Dynamic program slicing methods are very attractive for debugging because many statements can be ignored in the process of localizing a bug. Although language interoperability is a key concept in modern development platforms, current slicing techniques are still restricted to a single language. In this paper a cross-language dynamic program slicing technique is introduced for the .NET environment. The method is utilizing the CLR Debugging Services API, hence it can be applied to large multi-language applications.

Keywords

Program slicing, dynamic slicing, cross-language slicing, .NET Framework

1. INTRODUCTION

At the end of the seventies, when programming languages reached the level of maturity to directly support the construction of large software systems, an urging need for the extension of debugging, reverse engineering and software maintenance capabilities emerged. Science's answer to this challenge was program slicing [Tip95a]. The original goal of program slicing was to map mental abstractions made by programmers during debugging to a reduced set of statements in source code. As a consequence, it has always been highly desirable to integrate 'program slicers' with existing debugging environments.

A program slice contains all statements that might directly or indirectly affect the values of variables in a set `V` at a program location `p`. The pair `C=(p,V)` is usually referred to as a slicing criterion, and the contributing statements as the program slice with respect to slicing criterion `C`.

Since the original article of Weiser [Wei84a], many slightly different notions and algorithms have been developed to calculate program slices. As programming languages and existing technologies evolved, new features such as procedures, pointers, polymorphism, inter-process communication capabilities were also introduced, invalidating earlier definitions.

Weiser's original method is based on calculating consecutive sets of indirectly relevant statements based on control flow and data dependency analysis [Kri03a, Wei84a, Tip95a]. Later more advanced methods have been introduced by Ottenstein et al. calculating slices based on solving a reachability problem in the program dependency graph (PDG) [Ott84a]. A PDG is a directed graph with statements and control predicates in its vertices and edges corresponding to data and control dependences. A slicing criterion can be represented as a vertex in the PDG, and a slice with respect to this criterion contains all those vertices from which the vertex of interest can be reached.

What Weiser's and the PDG approach have in common is that they completely rely on *statically* available information to calculate program slices, therefore this method is called *static slicing*. Static slices have been specifically proposed for

***.NET Technologies'2005** conference proceedings,*
ISBN 80-86943-01-1
Copyright UNION Agency – Science Press, Plzen, Czech Republic

Supported by the Hungarian Ministry of Education under Grant FKFP 0018/2002 and Microsoft Hungary

maintenance and program understanding: one is able to use static slices to observe only parts of the program that may be relevant from one specific point of view [Bes01a]. However, making no assumptions about the program's input has a degrading effect on the precision of the obtained slice. Besides statements that actually affected the value of the variable under consideration, those that potentially did are also included in the slice. Although obtained with relatively small effort, the main disadvantage of slicing statically is usually the size of the slice.

While static slicing neglects actual program input, *dynamic slicing* [Agr91a, Bes01a, Tip95a, Zha03a] takes it into consideration. Static slicing can be simply thought of as a method which calculates statements possibly affecting the value of a variable of interest. The notion of dynamic slicing is much closer to running the program against a specific test case in a unit test: only dependences along a specific execution path are regarded. This approach implies that different occurrences of the same statement have to be considered. As a consequence, unlike a static (or classical) slicing criterion, a dynamic slicing criterion consists of a triple `(I, O, V)`, where `I` stands for program input, `O` is the occurrence of a statement and `V` is the set of variables under consideration.

As previously mentioned, a wide range of applications of program slicing have already been studied. But the highest potential is probably in debugging applications, where dynamic slicing is of great importance. One of the emerging concepts of modern real-world software systems is that they are built of a set of modules not necessarily written in the same programming language. During the whole lifecycle of such a system new features are added regularly as new modules, and existing legacy parts can also be refactored or integrated in such a way. Therefore, given a framework that directly supports cross-language programming, one has the capability to effectively slice real-world programs.

Introduced in 2001, designed with language interoperability as the key concept in mind, the .NET Framework is a platform where not only the widely studied inter-procedural but also 'cross-module' and 'cross-language' dynamic slicing techniques can be established. A module can be thought of as the equivalent of a .NET assembly. The term 'cross-language' means that each assembly might be composed of source code written in a different language. One of the most promising candidates for implementing a tool with this kind of capability is the .NET Debugging Services API.

Until now, the dynamic slicing community used the Java platform as its primary environment. Many interesting approaches have already been proposed, including slicing at bytecode level [Ume03a], bytecode transformation and JVM hacking.

However, there was no standard way to implement a debugger until Java Platform Debugger Architecture (JPDA) introduced in JDK 1.3. Besides having all primitives necessary to implement a debugger, JPDA also supports a number of debugging modes including in-process and out-of-process debugging. JPDA is an advanced API with many features similar to ones present in .NET. Since .NET was released more than five years after Java, we can rightly assume the presence of an additional set of features that could possibly support dynamic slicing.

In this paper we propose a pilot solution for cross-language dynamic slicing in the .NET Framework. Our main goal was to develop a dynamic slicing algorithm that takes advantage of the sophisticated debugging capabilities of the .NET platform. We also managed to implement a test application that is capable of dynamically slicing multi-module programs written in a C#-Visual Basic .NET mixed language environment.

2. OVERVIEW OF THE .NET ARCHITECTURE FROM THE POINT OF PROGRAM SLICING

In this section we give a brief overview of Microsoft's .NET architecture and explain why it is a perfect candidate for cross-language dynamic program slicing. We introduce the key concepts necessary to thoroughly understand the debugging capabilities of the framework.

.NET was originally designed to replace the classical Windows Programming Interface (WIN32 API), Component Object Model (COM) technology and its Distributed version (DCOM) and also to compete with the Java platform in the enterprise sector. As such, .NET offers all advantages of Java, along with language neutrality. All .NET languages use the same fully object-oriented runtime library. The philosophy behind this idea is the observation that it is easy to learn a new programming language; the hard part is when programmers are forced to learn many different class libraries and also legacy APIs. Using .NET, one is given the freedom to choose any of the 20+ supported languages and can get on with only one common library. This makes it easy to modify, transform or even integrate legacy systems.

However, some sophisticated machinery is needed to deliver these special features. To keep things simple, we propose a bottom-up overview of the architecture.

The Common Language Runtime (CLR) is the managed code lattice that everything else is built on. .NET uses just-in-time (JIT) compiled bytecode similar to HotSpot mechanism in Java.

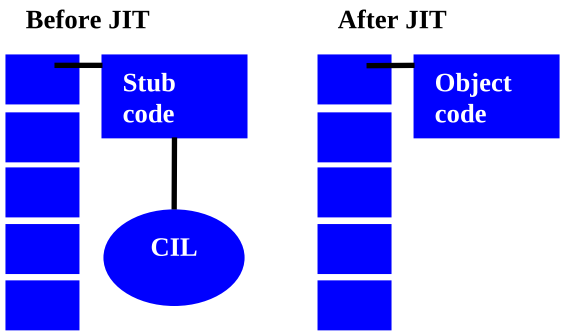


Figure 1: An assembly before and after jitting

Being also a fundamental part of the runtime's support for multi-language features, the Common Type System (CTS) provides basic value types, reference types, type safety, objects, interfaces, and delegates. It serves as a framework that helps the establishment of cross-language interoperability and type safety along with rapid execution capabilities.

The Common Language Specification (CLS) is the smallest subset of the CTS that all languages supported by the framework need to share. For example, two .NET languages can share values of non-CLS types but there will be languages which are unable to understand them.

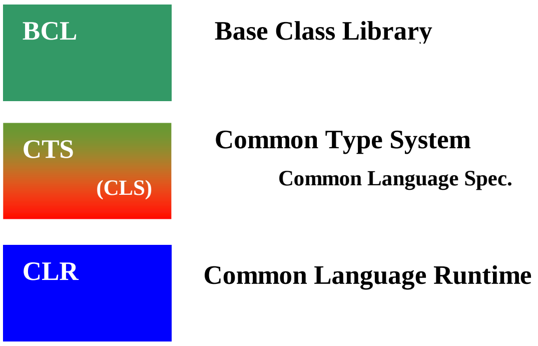


Figure 2: Overview of the .NET architecture

All .NET languages compile to an intermediate language code called Common Intermediate Language (CIL). The compiled code is organized into assemblies. Assemblies are portable executables - similar to dll's - with the important difference that assemblies are populated with .NET metadata and CIL code instead of normal native code. Figure 1 illustrates the way in which assemblies are jitted.

Figure 2 shows the details of the technology we have covered so far.

Companies tend to develop their specific solutions to a given problem, build custom libraries and user interfaces for their enterprise level applications. Modules are written separately in time and space, using different tools and compilers. In a later phase they are integrated, ideally in a seamless way. Unfortunately, in practice, this is rarely the case. A multi-language development platform supporting a large number of programming languages completed with a cross language and dynamic slicing capable debugger is a large step towards automatic – or at least towards seamless system integration.

In addition, with the help of cross-language program slicers programmers are able to identify bugs more precisely and at a much earlier stage. With the help of its sophisticated, carefully designed architecture and outstanding debugging capabilities, .NET is the platform that probably most closely matches the needs. In the case of program slicing, there is a two way symbiosis. Slicing improves software quality, and improved features of platforms like .NET may simplify slicing to a level where the power of its practical application appears.

However, it is not only the technical side that might benefit from such a framework. Microsoft is devoted to satisfying scientific needs as well with Rotor. Our approach focuses mainly on the possibilities of debugging from the scientific aspect. Debuggers are not toys, they are in fact serious tools in the hand of programmers. With the advanced features of .NET, a new generation of slicing capable debuggers is closer than ever before.

3. TECHNICAL OUTLOOK

In this section we give a brief overview of the basic architecture of JPDA widely used in the Java slicing community. The advanced architecture and the success of JPDA in slicing prompted us to introduce a similar approach in the .NET environment. We intend to show how .NET Debugging Services – the .NET counterpart of JPDA - can be used to generate call trace of the program being sliced.

JPDA is a multi-layer architecture dedicated to the direct support of debugger application development. Since JPDA fits in the philosophy of Java, debuggers based on this architecture are intended to run on a variety of physical platforms, virtual machines and also JDKs.

The main three layer of JPDA are:

1. Java Virtual Machine Debug Interface (JVMDI): all debugging services provided by the VM
2. Java Debug Wire Protocol (JDWP): specifies communication standards between the debugger and the process being debugged
3. Java Debug Interface (JDI): the top level interface for debugger developers.

JVMDI is the lowest layer of JPDA. It exposes both state inspection and controlling capabilities of applications running in a virtual machine to debugger developers. Basically, JVMDI is an event-driven interface. However, it has also indirect controlling capabilities totally independent of events. Default JVMDI clients are in-process, that is they run in the same virtual machine as the application that is being debugged. On the other hand, the framework also contains higher-level, out-of-process debugger interfaces.

JDWP is a communication protocol between the virtual machine being debugged and the debugger process. This protocol ensures that a single debugger is able to work either locally or (in a distributed way) on a remote computer. A very important aspect of JDWP is it independence of transport mechanisms. Every different JDWP implementation might employ different transport techniques through a simple API.

JDI is the highest level JPDA interface providing information that is of great importance in case of debuggers and also other tools that need access to the running state of a virtual machine.

In the Microsoft world, with the release of .NET, a new Debugging API and scripting strategy has also been introduced. Script engines can now compile or interpret code for the Microsoft Common Language Runtime (CLR) instead of integrating debugging capabilities directly into applications through Active Scripting [Pell]. .NET Debugging Services is not only able to debug every code compiled to IL written in any high level language, but it also provides debugging capabilities for all modern languages.

The CLR supports two types of debugging modes: in-process and out-of-process. In-process debuggers are used for inspecting the run-time state of an application and for collecting profiling information. These kinds of debuggers do not have the ability to control the process or handle events like stepping, breakpoints, etc.

Out-of-process debuggers run in a separately process providing common debugger functionality.

The CLR Debugging Services are implemented as a set of some 70+ COM interfaces, which include the *design-time application*, the *symbol manager*, the *publisher* and the *profiler*.

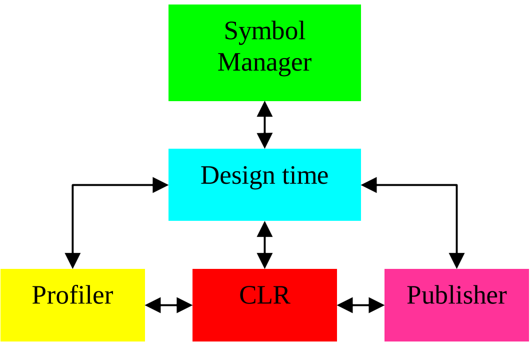


Figure 3: CLR Debugging architecture

The *design-time interface* is responsible for handling debugging events. It is implemented separated from the CLR while the host application must reside in a different process. The application is interpreted by a script and has a separate thread for receiving debugger events that run in the context of the debugged application. When a debug event occurs (assembly loaded, thread started, breakpoint reached, etc.) the application halts and the debugger thread notifies the debugging service through callback functions.

The *symbol manager* is responsible for interpreting the program database (PDB) files that contain data used to describe code for the modules being executed. The debugger also uses assembly metadata that also holds useful information from the point of debugging. The PDB files contain debugging information and are generated only when the compiler is explicitly forced to do so. Besides enabling the unique identification of program elements like classes, functions, variables and statements, the metadata and the program database can also be used to retrieve their original position in the source code.

The *publisher* is responsible for enumerating all running managed processes in the system.

The *profiler* tracks application performance and resources used by running managed processes.

The CLR Debugging Services API called *ICorDebug* [Stall] is implemented by COM interfaces. It can be directly reached from managed or unmanaged code but there are also higher level managed wrapper classes used by MDbg [Stall]. Using these interfaces we can start a process for debugging and register our managed or unmanaged callback functions. As mentioned earlier, querying run-time information of program elements is another important application.

We generated the call trace of our programs using the CLR debugger. First we set a breakpoint to the entry of our application and we stepped along until the end. The step (or step in) debugging operation goes along sequence points in the original source code. Sequence points which can be identified using metadata and the program database divide the statements in high-level languages. We also used ICorDebug to query the function call stack at every step.

ICorDebug has not been standardized yet and it is not likely to be. According to Mike Stall [Stall] it makes more sense to standardize the compiler's output (metadata, symbols, IL format). We have also studied the other two significant .NET implementations namely Microsoft's SSCLI (Rotor) and Mono sponsored by Novell. Rotor has the same debugging architecture as the Microsoft .NET Framework so it would be easy to compile and run our existing tracer application on that platform. On the other hand, Mono developers decided against implementing the debugging API provided by the .NET CLR and Rotor and have their own debugging mechanism. Fortunately, the module generating call trace accounts for only a very small part of our dynamic slicing framework so it would take relatively small effort to port it to Mono.

4. ARCHITECTURE & ALGORITHM

In this section we will review the architecture (Fig. 4) of our dynamic slicing framework. It consists of two phases called *Phase 1* and *Phase 2*. While Phase 1 executes mainly preprocessing steps, Phase 2 runs the slicing algorithm. The whole framework was developed and compiled using Microsoft Visual Studio 2005 beta.

The current implementation of our dynamic slicing algorithm, that is capable of processing source code only line-by-line, makes the first step of *Phase 1* - 'beautification' - necessary. Beautification is a preprocessing step that enables the debugger to generate a call trace that is the input of our dynamic slicing algorithm. Beautification requires a language-specific parser transforming the original code to an equivalent version split along sequence points. As a result of the beautification step the source code lines can be directly mapped to sequence points that the debugger is capable of stepping along. As a consequence, the mapping between lines and sequence points makes it possible to use the output of the debugger as the direct input of the dynamic slicing algorithm.

Since the CLR Debugger is language-independent and parsers can be developed for any language, it is possible to generate slices that span across multiple assemblies compiled from different languages.

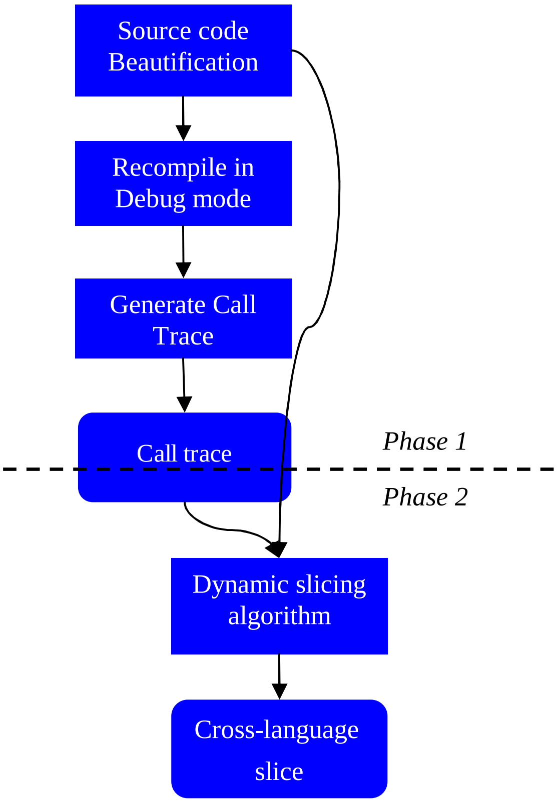


Figure 4: Architecture

In case of C#, we compile the beautified source files by calling the C# compiler csc.exe with the /debug+ switch to generate debugging output. The last step of Phase 1 is the building of the call trace which is written to a plain text file. We trace information of every single statement reached during the execution of our program using .NET Debugging Services API. As we have already mentioned, the *ICorDebugStepper* interface is used to step along the application. At each step a triple of data is stored, namely:

1. The name of the source file name we are in
2. The exact line number in the source file where the statement of interest resides
3. The state of the call stack at that point

Each element of the triple holds meaningful information for our dynamic slicing algorithm. Since the analyzed application can be built-up of multiple assemblies (and multiple source files), therefore the correct place including the source file name and exact line number always have to be recorded. The call stack is used for tracking function calls.

Phase 2 first loads the call trace file produced in Phase 1. A typical call trace can be seen in Listing 1.

Although in a real application we store fully qualified names, for the sake of clarity we have used abbreviations in Listing 1, so M stands for MainNameSpace.MainClass.Main, R for MainNameSpace.MainClass.RecursiveProdSum, A for OtherModule.Functions.Add and P for Prod.

```
idx01: MainClass.cs 10 M
idx02: MainClass.cs 11 M
idx03: MainClass.cs 12 M
idx04: MainClass.cs 13 M
idx05: MainClass.cs 14 M
idx06: MainClass.cs 20 M,R
idx07: MainClass.cs 22 M,R
idx08: Functions.cs 10 M,R,A
idx09: Functions.cs 11 M,R,A
idx10: MainClass.cs 23 M,R
idx11: Functions.cs 15 M,R,P
idx12: Functions.cs 16 M,R,P
idx13: MainClass.cs 24 M,R
idx14: MainClass.cs 25 M,R
idx15: MainClass.cs 20 M,R,R
…
```

Listing 1: Call Trace

A screenshot of the framework with source code corresponding to the call trace in Listing 1 can be seen in Figure 6.

The next step is to parse traced source files for every assembly in the program. We use here the same parser as in the beautification step. Being similar to existing dynamic slicing algorithms in this aspect [Bes01a, Xu01a, Zha03a], our approach also necessitates storing referenced and defined variables at every statement. The main task of the parser is to collect referenced and defined variables at every statement. This is illustrated in the following code fragment.

```
 1 int n = askUser();
 2 int i = 0;
 3 int sum = 0;
 4 int prod = 1;
 5 while (i < n)
 6 {
 7  sum += i;
 8  prod *= i;
 9  i++;
10 }
11 Console.WriteLine(sum);
```

Listing 2: Simple C# code fragment

Line 2 defines variable i, line 5 references i and n, line 7 defines sum and references sum and i, line 11 references sum.

While parsing source files, a *Control Dependence Graph (CDG)* [Kri03a] is also created. Control dependence describes the ability of a program statement to affect the execution of another program statement. If node m is control dependent on node n it means that there is an edge from n to m. Figure 5 illustrates the CDG of the code fragment given in Listing 2.

```
loopcond←∅

varstore←∅


foreach var∈{slicing_crit_vars} loop
  varstore←varstore∪(var,Ref)
end foreach

foreach stmt in {backward call trace} do
  if stmt is Assignment then
    found:= false
    foreach var∈{stmt.definedvars} do
      if (var,Ref)∈varstore then
        varstore[(var,Ref)]←(var,Def)
        found:= true
      end if
    end foreach
    if found then
      slice:=slice∪{stmt}
      addToVarStoreAndLoopCond(stmt)
    end if
  else
  if stmt is control statement then
    if stmt∈ loopcond then
      slice←slice∪{stmt}
      addToVarStoreAndLoopCond(stmt)
    end if
  end if
end loop

proc addToVarStoreAndLoopCond(stmt)
  foreach var∈{stmt.referencedvars} do
    varstore←varstore∪(var,Ref)
  end foreach

  foreach parstmt in {stmt.parents} do
      loopcond←loopcond∪parstmt
  end foreach
end proc
```

Listing 3: Intra-procedural version of our dynamic slicing algorithm

For example, nodes 1, 2, 3, 4, 5, 11 and 7, 8, 9 are neighbors; 7, 8, 9 are control dependent on 5.

The call trace for our example program is the following in regular expression style: "1,2,3,4(,5,7,8,9){n},5,11". The slicing criterion is (<n=2>, 11^1, {sum}).

According to the definition given in Section 1, <n=2> is the current program input, 11^1 denotes the first

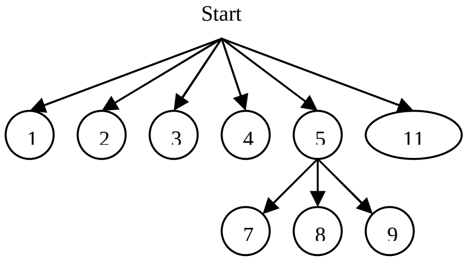


Figure 5: Control Dependence Graph

occurrence of the statement in source code line 11 in the call trace and `sum` is the only variable of interest. In other words, we are interested in statements that affect the value of variable `sum` when we reach the 11th line for the first time with n=2 being the input of the program.

At this point we have all information necessary to develop our backward dynamic slicing algorithm. First we will show it in an *intra-procedural* form then extend it to the more interesting *inter-procedural* version.

We have a set (called *`varstore`*) whose elements are (Variable, `Action`) pairs where `Action` can be either Def or Ref. `Varstore` is responsible for storing the *last* `Action` for every variable of interest. Def denotes variable definition; similarly Ref denotes referencing that variable.

When the algorithm starts, `varstore` contains all variables of interest with Ref `Action`. For the previous example: (sum, Ref). When a variable with Ref action is encountered on the left side of an *assignment*, the line number is added to the dynamic slice (if not already in) and the variable's Ref `Action` is changed to Def. (We are not interested in assignments defining a variable with Def action, because the earlier definition would be killed anyway.) The `Action` of referenced variables with Def `Action` is changed to Ref. Referenced variables not already in `varstore` are added with Ref `Action`. (For example, encountering `i++` would first change the `Action` of `i` to Def and then Ref).

After processing a statement we always add its parent according to the CDG to another set called *`loopcond`*. `Loopcond` stores those control flow statements (loop or condition) that have to be added to the slice during the first visit. When a control flow statement is encountered, we check whether it is in `loopcond`. In this case we process it similar to assignments (set Ref variables, add parents to `loopcond`, increase dynamic slice).

The outcome of the algorithm run against code fragment in Listing 2 is shown in Table 1.

The algorithm is linear in the number of lines in the call trace; memory usage is also linear with respect to the number of variables in `varstore`.

trace	Varstore	loop-cond	Slice
11	(sum,Ref)	-	-
5	(sum,Ref)	-	-
9	(sum,Ref)	5	-
8	(sum,Ref)	5	-
7	(sum,Ref),(i,Ref)	5	7
5	(sum,Ref),(i,Ref),(n,Ref)	-	5,7
9	(sum,Ref),(i,Ref),(n,Ref)	5	5,7,9
8	(sum,Ref),(i,Ref),(n,Ref)	5	5,7,9
7	(sum,Ref),(i,Ref),(n,Ref)	5	5,7,9
5	(sum,Ref),(i,Ref),(n,Ref)	-	5,7,9
4	(sum,Ref),(i,Ref),(n,Ref)	-	5,7,9
3	(sum,Def),(i,Ref),(n,Ref)	-	3,5,7,9
2	(sum,Def),(i,Def),(n,Ref)	-	2,3,5,7,9
1	(sum,Def),(i,Def),(n,Def)	-	1,2,3,5,7,9

Table 1: Algorithm example

The algorithm starts exactly the same way in the inter-procedural case as the previously introduced intra-procedural version. However, when the last line of a function (eg. in Listing 1 Functions.cs line 11) is reached, the line from where the function was called have to be identified even in the case of multiple or recursive calls (eg. in Listing 1 MainClass.cs line 22). Also, all local variables that are parameters of the called function have to be localized.

The calling statement can be found in linear time in the call trace so the algorithm would become quadratic. However, some preprocessing can be done to preserve the linearity of our algorithm. A unique ID is given to every function call. Note that the blocks of the same ID-runs do not have to be continuous (eg. for Listing 1 this would be 1,1,1,1,1,2,2,3,3,2,4,4,2,2,5,…). At a given block of IDs the ending index of the previous block of the same IDs can be stored (eg. for statement at `idx10` we store `idx7`, for `idx13` store `idx10` as shown in Listing 1). So we can find the calling statement in one step even for multiple or recursive calls.

In order to achieve constant-cost retrieval of the index that marks the end of the previous block with the same IDs, an *indexing data structure* should be created and populated in a preprocessing step. At this point we are aware of the statement that calls the function and can further investigate the in/out (`ref` in C#) and out (`out` in C#) actual parameters.

The algorithm selects parameter variables of the caller function with Ref `Action` in `varstore` (we call them *formal parameters of interest*). If there is no variable satisfying this criterion, we can safely disregard the whole function.

```
Function: doSliceFunction(Context context, int funcend)
context.CalculateStartingVarStore()
funcID:= -1;
while actLine > funcEnd do
begin
  TraceLine trace = callTrace[actLine]
  if funcID = -1 then funcID:= trace.FuncID

  //when a new function reached
  if trace.FuncID <> funcID then
  begin
    callPos:= rle[actRLELine].PrevBlockEnd
    actRLELine:= actRLELine - 1
    TraceLine traceMI:= callTrace[callPos]
    MethodInvoke mi:=
          source[traceMI.src].Statement[callPos]
    actualParamsOut:=
      mi.Outputs.SelectReferenceds(context.VarStore)
    formalParamsOut:= mi.Actual2Formal(actualParamsOut)
    Context newContext:= new Context(formalParamsOut)
    doSliceFunction(newContext, callPos)
    formalParamsIn:= newContext.SelectReferenceds(
        mi.Parameters)
    if formalParamsIn.Count > 0 then
    begin
      actualParamsIn:= mi.Formal2Actual(formalParamsIn)
      context.VarStore.InsertThemAsRef(actualParamsIn)
      slice←slice ∪ {mi}
      foreach parstmt in {stmt.parents} do
        context.loopcond←context.loopcond ∪ parstmt
      end foreach
    end if
    actRLELine:= actRLELine – 1
    actLine:= actLine - 1
    continue
  end if

  //normal statement
  Statement stmt:=
          source[trace.src].Statement[trace.line]
  if stmt is Assignment then
    found:=false
    foreach var∈ {stmt.definedvars} do
      if (var,Ref)∈ context.VarStore then
        context.VarStore[(var,Ref)]←(var,Def)
        found:=true
      end if
    end foreach
    if found then
      slice:=slice ∪ {stmt}
      addToVarStoreAndLoopCond(stmt)
    end if
  else
  if stmt is control statement then
    if stmt∈ context.loopcond then
      slice←slice ∪ {stmt}
      addToVarStoreAndLoopCond(stmt)
    end if
  end if
  actLine--
end while
```

Listing 4: Inter-procedural slicing algorithm

Since functions can be identified based on the signature of the calling statement, formal parameters can be identified according to their order. Now we can recursively call our dynamic slicing algorithm by setting up a new `varstore` with all formal parameters of interest with Ref `Action`. When the algorithm returns to the caller we can identify all formal input parameters (nothing or ref in C#) *referenced from the generated slice* by checking the `varstore` of the called function and determine their actual parameter pairs. We consider them as referenced variables from the caller's point of view. So they are added to the `varstore` with Ref `Action` or their `Action` value is changed to Ref if already in `varstore`. We modify `loopcond` in the exactly similar way as in the case of assignments and of course also add the function call to the slice.

It can be seen that we store unique `varstore` and `loopcond` information for every function call. Listing 6 shows the pseudo code of the inter-procedural version of our dynamic slicing algorithm. As its name suggests, variable `callTrace` stores information generated with the help of .NET Debugging Services. The algorithm walks from the end to the beginning of the call trace. Index `actLine` decreases at every step of the algorithm. Variable `funcEnd` stores the location where the currently processed function is called. If this point is reached we go back to the caller. The statements are identified by source files (which can belong to different modules) and the line number in the source file. When the algorithm detects that the execution passed the last line of a method, the source file and line number (`funcEnd`) are identified where the invocation of this method is performed. Actual output parameters referenced according to `varstore` are looked up and their formal output parameter pairs are matched. Afterwards, the dynamic slicing algorithm is called recursively.

Returning from the recursion, the referenced formal input parameters and their actual counterparts are also identified. They are added to `varstore` and the algorithm continues.

Function `addToVarStoreAndLoopCond` is almost the same presented in Listing 3 except for that `loopcond` and `varStore` are referenced by context.

5. IMPLEMENTATION

In the screen shot shown in Figure 6 we used slicing criterion (`<n=42>`, 15[1], `{sum}`). The example contains two files from different assemblies (MainClass is in the main module and Functions class which is used in the main module is located in another module).

In order to test the algorithm proposed earlier, we have implemented a pilot application that is capable of slicing programs that satisfy certain restrictions. These restrictions imply that the source code might contain only static functions with arbitrary program constructions (assignment, condition, loop, method invocation). The program can be built of multiple modules (assemblies) each containing multiple source files.

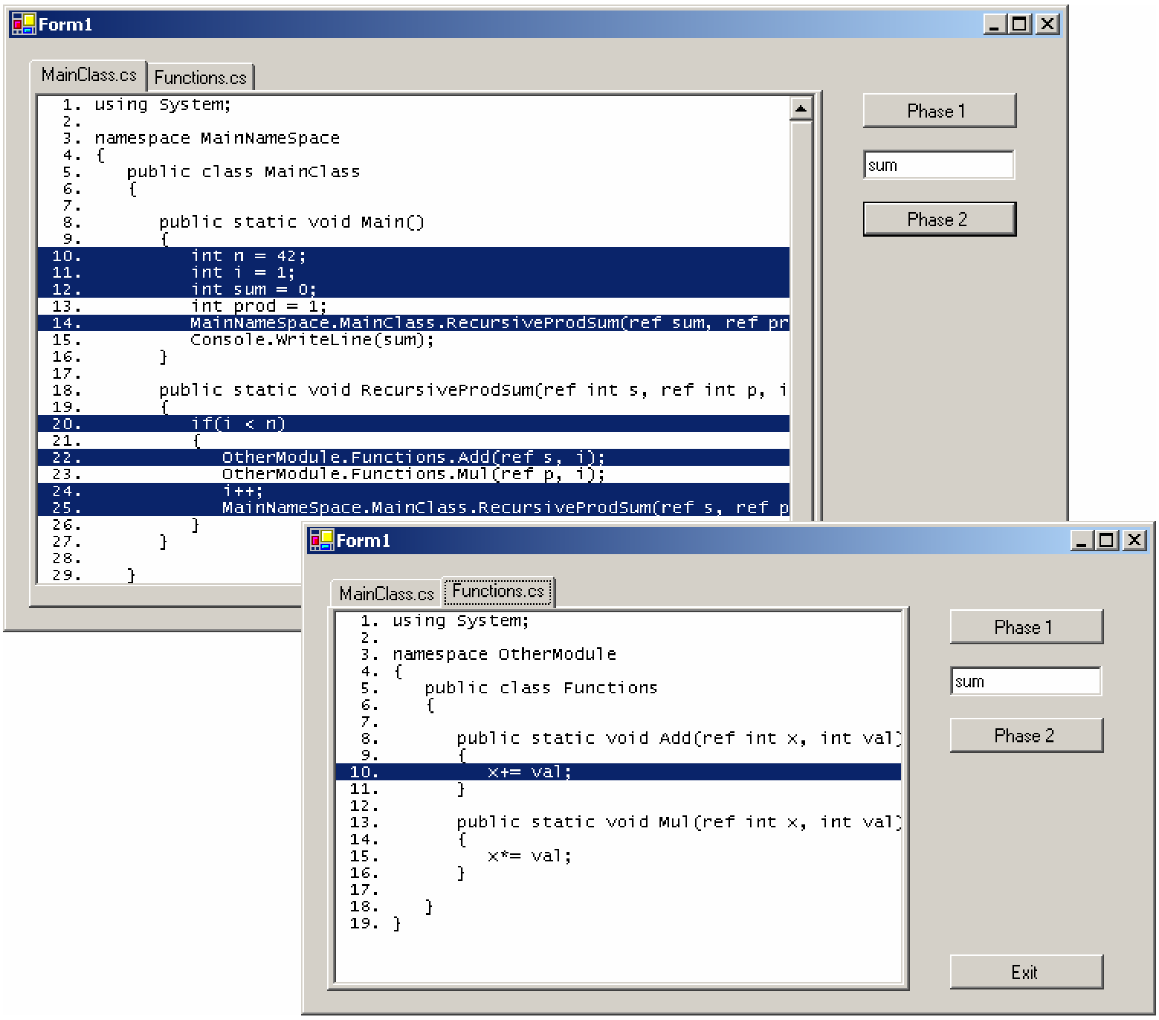
Since the CLR Debugger is language-independent and parsers can be developed for any language, it is possible to generate slices that cover multiple assemblies compiled from different languages. Unfortunately the only parser we have is for C#.

We used an earlier version of Marcel Debreuil's C# source code parser library which employs the ANTLR parser generator. We compiled our algorithm using Microsoft Visual Studio 2005 beta codenamed Whidbey.

6. CONCLUSION AND FURTHER WORK

In this paper we have shown how to utilize the .NET Debugging Services API in dynamic program slicing. Motivated by the Java Platform Debugger Architecture, our pilot solution can be effectively used to investigate dynamic dependences among modules compiled from any CLS-compliant language. We have also shown that by directly supporting cross-language programming, the .NET Framework offers significant surplus over Java.

.NET-languages, mainly C#, VB.NET and managed C++ have some very noteworthy elements such as



```
using System;

namespace MainNameSpace
{
    public class MainClass
    {

        public static void Main()
        {
            int n = 42;
            int i = 1;
            int sum = 0;
            int prod = 1;
            MainNameSpace.MainClass.RecursiveProdSum(ref sum, ref pr
            Console.WriteLine(sum);
        }

        public static void RecursiveProdSum(ref int s, ref int p, i
        {
            if(i < n)
            {
                OtherModule.Functions.Add(ref s, i);
                OtherModule.Functions.Mul(ref p, i);
                i++;
                MainNameSpace.MainClass.RecursiveProdSum(ref s, ref p
            }
        }

    }
```

```
using System;

namespace OtherModule
{
    public class Functions
    {

        public static void Add(ref int x, int val)
        {
            x+= val;
        }

        public static void Mul(ref int x, int val)
        {
            x*= val;
        }

    }
}
```

Figure 6: Example run of our slicing framework

delegates, the foreach loop, different kinds of parameter passing methods and the lock statement which justify further research related to both static and dynamic program analysis.

C# language and .NET Framework are evolving quickly. In Microsoft .NET Framework version 2.0 we intend to investigate generics, anonymous methods, partial types, yield keyword, nullable types and also some functional language implementations like Scheme [Bre04a] and Clean [Her04a].

REFERENCES

[Agr91a] H. Agrawal and J. R. Horgan. Dynamic program slicing. In SIGPLAN Notices No. 6, pages 246-256, 1990.

[Bes01a] Á. Beszédes, T. Gergely, Zs. M. Szabó, J. Csirik, T. Gyimóthy. Dynamic slicing method for maintenance of large C programs, CSMR 2001, pages 105-113.

[Bre04a] Bres,Y., Serpette,P., Serrano,M. et al. - Compiling Scheme programs to the .NET Common Intermediate Language, 2nd International Workshop on .NET Technologies, May 2004

[Her04a] Z. Hernyák, Z. Horváth, V. Zsók. Design of Language Elements for Dynamic Distributed Computation of Clean Expressions on Clusters. Submitted to TFP 2004 Fifth Symposium on Trends in Functional Programming, Ludwig-Maximilians University, Munich, Germany, 2004.

[Hor90a] S. B. Horwitz, T. W Reps, D. Binkley. Inter-procedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems, 12(1): 26-60, January 1990.

[Kri03a] J. Krinke, Advanced Slicing of Sequential and Concurrent Programs, PhD Thesis, Universität Passau, April 2003

[Mar03a] K. Maruyama, M. Terada, Timestamp Based Execution Control for C and Java Programs, AADEBUG, 2003

[Oha01a] F. Ohata, K. Hirose, M. Fujii, K. Inouse. A slicing method for object-oriented programs using lightweight dynamic information. In Proc. of the 8th Asia-Pacific Software Engineering Conference, 2001.

[Ott84a] K. J. Ottenstein, L. M. Ottenstein. The program dependence graph in software development environment. ACM SIGPLAN Notices volume 19(5), pages 177-184, 1984.

[Pel02a] M. Pellegrino. Improve Your Understanding of .NET Internals by Building a Debugger for Managed Code. MSDN Magazine, issue November 2002. http://msdn.microsoft.com/msdnmag/issues/02/11/clrdebugging/

[Rep94a] T. Reps, S. Horwiz, M. Sagiv, G. Rosay. Speeding up slicing. ACM SIGSOFT Software Engineering Notices 19, pages 11-20.

[Stall] Mike Stall's .NET Debugging Blog, http://blogs.msdn.com/jmstall/, 2004-2005

[Tip95a] F. Tip, A survey of program slicing techniques. Journal of Programming Languages, 3(3):121-189, Sept. 1995.

[Ume03a] F. Umemori, K. Konda, R. Yokomori, K. Inoue, Design and Implementation of Bytecode-based Java Slicing System, SCAM 2003

[Wei84a] M. Weiser. Program Slicing. IEEE Transactions on Software Engineering. SE-10(4):352-357, 1984.

[Xu01a] B. Xu, Z. Chen. Dependence Analysis for Recursive Java Programs. In SIGPLAN Notices No. 12, Pages 70-76.

[Zha03a] X. Zhang, R. Gupta, Y. Zhang. Precise dynamic slicing algorithms. Proc. International Conference on Software Engineering, pages 319-329, 2003.