# Echo-Emu Documentation

***Release latest***

**Jul 19, 2023**

# General

This is the documentation of the Echo project. Echo is a generic, static analysis, symbolic execution and emulation framework, that aims to help out with binary code analysis for a variety of platforms.

CHAPTER 1

Table of Contents:

## 1.1 Introduction

### 1.1.1 What is Project Echo? (short version)

Echo is a high-level static and dynamic analysis framework, providing models and algorithms for analysing control and data flow of low-level assembly-like code, as well as symbolic execution and emulation engines for a variety of platforms.

### 1.1.2 OK but now in normal English (long version)

In software reverse engineering, we often try to figure out what a particular piece of code does. We generally use static analysis tools such as disassemblers, that not only decode instructions but also perform some analysis on them, such as the construction of control and data flow graphs.

Over the past years, a lot of different architectures have been developed, each with their own instruction sets. Simultanuously, lots of static analysis libraries have been developed to decode and analyse code targeting these platforms.

The problem is that most of these libraries only target one single platform, and are often incompatible with each other. As a result, lots of the standard analysis algorithms (such as the aforementioned control and data flow analysis) has to be adapted or completely recoded to accomodate for all the differences in implementation. This especially becomes an issue when we look at really obscure instruction sets, such as the ones provided by old processors, or virtual machines used by obfuscators. Writing static analysis tools for these kinds of platforms with the same kind of capabilities as the ones for more well-known platforms, would require lots of time and effort, even though the algorithms are more or less the same.

Project Echo tries to solve this by abstracting away the underlying instruction set through a set of interfaces that try to capture all similarities between different platforms. A developer could then "simply" implement these interfaces and describe *what* a specific platform looks like, and not having to worry about how to implement complicated analysis algorithms for their instruction set.

### 1.1.3 Goals

The goals of Project Echo include (but are not limited to):

- To provide a model for program code of a variety of platforms.
- To provide a framework that models the state of a program, including variables, stack, and memory.
- To provide the ability to analyse the control and data flow within chunks of code.
- To provide the means to emulate (parts of) the program code, and simplify expressions using symbolic execution and SAT solvers.
- To provide the means to export control flow graphs to various file formats.

## 1.2 General Project Structure

The Echo project mainly consists of two parts. The core libraries that is shared amongst all platforms, and the various back-end libraries that implement the interfaces to describe various platforms.

### 1.2.1 Core

The core libraries expose the base interfaces that all platforms should implement, as well as the analysis algorithms that work on these models.

- **Echo.Core:** The core package, containing base models and interfaces for graph-like structures and program code.
- **Echo.ControlFlow:** The package providing models and algorithms for anything related to control flow analysis. This includes extracting control flow graphs from instruction streams, as well as serialization to blocks.
- **Echo.DataFlow:** The package providing models and algorithms for anything related to data flow analyis. This includes extracting data flow graphs, symbolic values, program slicing and finding dependency instructions.
- **Echo.Concrete:** The package providing base models for emulator packages. This includes the implementation of various concrete emulated values, as well as interfaces and base models for code emulators and interpreters.

### 1.2.2 Platforms

Different platforms have different features and instruction sets. Therefore, for Echo to do analysis on code targeting such a platform, it needs to know certain properties about what the code looks like, and how a program evolves over time.

A platform typically implements at least the following interfaces from the core libraries:

- **Code model interfaces:**
  - `IInstructionSetArchitecture<TInstruction>`: An interface representing an instruction set architecture (ISA) and is used to extract various kinds of information from a single instruction modelled by `TInstruction`.
  - `IVariable`: An interface describing a single variable.
- **Emulation interfaces:**
  - `IProgramState`: An interface describing what the current state of a program at a given point in time might look like. This includes (global) variables, the stack and memory state.

## 1.3 Road Map

Current nice-to-have features include:

### 1.3.1 Control Flow Analysis

- [x] Static CFG builder
- [x] Symbolic CFG builder
- [x] Traversal algorithms
- [x] Dominator analysis
- [x] Dot serialization
- [x] Block serialization
- [x] Method body serialization

### 1.3.2 Symbolic Execution

- [x] Generic symbolic execution engine base
- [x] CIL AsmResolver symbolic execution back-end
- [x] Data flow analysis
- [ ] Generic AST builder
- [ ] Static single assignment transformer
- [ ] Variable inlining

### 1.3.3 Concrete Emulation

- [ ] CIL emulator
- [ ] x86 emulator
- [ ] JVM emulator

## 1.4 The Basics

Graph-like structure form an integral part of Echo.

### 1.4.1 Graph interfaces

and are modelled in Echo using the interfaces found in the `Echo.Core.Graphing` namespace. The base interfaces include:

- `IGraph`: A container for a collection of nodes and edges.
- `INode`: A single node.
- `IEdge`: An edge between two nodes.

- `ISubGraph`: A collection of nodes and edges within a graph.

---

**Note:** It is recommended for any package that implements some kind of graph-like structure (including trees), to use the `INode`, `IEdge` and `IGraph` interfaces. This allows for leveraging all kinds of graph-related algorithms, such as traversals, sortings and serialization methods.

---

### 1.4.2 Inspecting the structure of graphs

The contents of the graph can be accessed using the `GetNodes()` and `GetEdges()` methods:

```
IGraph graph = ...;

foreach (INode node in graph.GetNodes())
    Console.WriteLine(node.Id);

foreach (IEdge edge in graph.GetEdges())
    Console.WriteLine($"{edge.Origin.Id} -> {edge.Target.Id}");
```

Every node is assigned a unique *Id*. Individual nodes can be obtained from the graph using the `GetNodeById(long)` method. Depending on the implementation of the graph, this identifier might have different meanings. For example, in control flow graphs, the identifiers of each node is the starting offset of the basic block.

```
IGraph graph = ...;
INode node = graph.GetNodeById(10);
```

Every node in a graph consists of at least the following methods that can be used to navigate through the graph:

- `IEnumerable<IEdge> GetIncomingEdges()`: Gets a collection of all incoming edges i.e. edges that target this node.
- `IEnumerable<IEdge> GetOutgoingEdges()`: Gets a collection of all outgoing edges i.e. edges originating from this node.
- `IEnumerable<INode> GetPredecessors()`: Gets a collection of nodes that precede this node i.e. all end points of each incoming edge.
- `IEnumerable<INode> GetSuccessors()`: Gets a collection of nodes that succeed this node i.e. all end points of each outgoing edge.

### 1.4.3 Example implementations

Various parts of Echo implement the graph interfaces:

- Control Flow Graphs (CFGs)
- Dominator trees
- Data Flow Graphs (DFGs)

### 1.4.4 Exporting Graphs

Echo also has the capability to export graphs:

- Exporting graphs to dot files

## 1.5 Dot file serialization

Dot files is a standard file format to store graphs in. They can also be used to visualize graphs using tools like GraphViz (Online version: http://webgraphviz.com/).

### 1.5.1 The basics

To export a graph constructed by Echo to the dot file format, use the *DotWriter* class.

First make sure you have a `TextWriter` instance, such as a `StringWriter`, a `StreamWriter` or `Console.Out`:

```
TextWriter writer = new StringWriter();
```

Then create a new dot writer:

```
var dotWriter = new DotWriter(writer);
```

Finally, write the graph:

```
IGraph graph = ...
dotWriter.Write(graph);
```

### 1.5.2 Graph adorners

Nodes and edges can be decorated with additional styles. This is done through the `IDotNodeAdorner` and `IDotEdgeAdorner` interfaces, which can be passed onto an instance of a `DotWriter` class.

Echo defines a few default adorners:

- `HexLabelNodeIdentifier`: Used to put labels on the nodes in hexadecimal format.

**Control flow graph adorners:**

- `ControlFlowNodeAdorner`: Used to include the contents of the embedded basic block in a node of a control flow graph visualization.
- `ControlFlowEdgeAdorner`: Used to add colour codes to an edge based on the type of edge.

**Data flow graph adorners:**

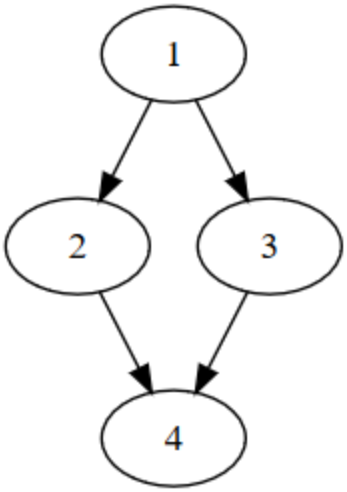
- `DataFlowNodeAdorner`: Used to include the instruction of the node of a data flow graph visualization.
- `DataFlowEdgeAdorner`: Used to add colour codes to an edge based on the type of data dependency.

## 1.6 The Basics

### 1.6.1 What are Control Flow Graphs?

A control flow graph (CFG) is a graph where each node corresponds to one basic block of code, and each edge represents a possible control flow transfer. This way, a control flow graph tries to encode all possible execution paths in a chunk of code, method or entire program.

The figure below depicts a control flow graph of a very simple if-statement:




### 1.6.2 Control Flow Graph Models

There are three main classes that are used to model CFGs. Given a `TInstruction` type, representing the type of instructions to store in the CFG, Echo defines the following classes:

- `ControlFlowNode<TInstruction>`: A single node containing a basic block of instructions.
- `ControlFlowEdge<TInstruction>`: A control flow transfer between two nodes.
- `ControlFlowGraph<TInstruction>`: A collection of control flow nodes and edges.

These classes implement the `INode`, `IEdge` and `IGraph` interfaces, and work therefore with all kinds of generic graph algorithms and export features.

CFGs can also be subdivided into multiple regions. These are represented using the `IControlFlowRegion<TInstruction>` interface, and are accessible through the `Regions` property of the `ControlFlowGraph<Tinstruction>` class. Echo provides the following base implementations:

- `BasicControlFlowRegion<TInstruction>`: A basic collection of nodes.
- `ExceptionHandlerRegion<TInstruction>`: A region representing an exception handler, consisting of a protected region and a collection of handler regions.

### 1.6.3 Inspecting basic blocks

A single `ControlFlowNode<TInstruction>` contains a single basic block of code. A basic block is a sequence of contiguous instructions that contains no jumps or labels. To access the basic block stored in the node, use the `Contents` property, which is of type `BasicBlock<TInstruction>`:

```
long offset = ...
var node = cfg.GetNodeByOffset(offset);
var block = node.Contents;

// Iterate over all instructions within the block.
foreach (var instruction in block.Instructions)
    Console.WriteLine(instruction);

// Get immedaitely the header/footer of the basic block.
```

(continues on next page)

(continued from previous page)

```
var firstInstruction = block.Header;
var lastInstruction = block.Footer;
```

### 1.6.4 Inspecting incoming and outgoing edges

Control flow nodes might transfer control to another block within the graph. To model this, instances of `ControlFlowNode<TInstruction>` define various properties that allow access to edges that are incident to this node.

There are three types of edges in a control flow graph:

- `FallthroughEdge`: The default edge that is taken when no other edge is taken.
- `ConditionalEdges`: Edges that are only taken when a particular condition is met.
- `AbnormalEdges`: Edges that are taken in the case of a rare event (such as exceptions).

These three properties are fully mutable, and can therefore be used to add or remove any edge within the graph to change the flow of a program.

However, the recommended way to add new edges is using the `ConnectWith` method:

```
var node1 = cfg.GetNodeByOffset(offset1);
var node2 = cfg.GetNodeByOffset(offset2);
var node3 = cfg.GetNodeByOffset(offset3);

// Adds a fallthrough edge from node1 to node2.
node1.ConnectWith(node2);
// Adds a conditional edge from node1 to node2.
node1.ConnectWith(node3, ControlFlowEdgeType.Conditional);
```

Updating any of the three properties will automatically update the return value of the‘‘GetIncomingEdges()‘‘ method of the target node.

```
var node1 = cfg.GetNodeByOffset(offset1);
var node2 = cfg.GetNodeByOffset(offset2);

// Adds a fallthrough edge from node1 to node2.
node1.ConnectWith(node2);

var incomingEdge = node2.GetIncomingEdges().First();
```

## 1.7 Static CFG Construction

The easiest, and probably most efficient way to construct a control flow graph using Echo is using a static control flow graph builder. This is generally good enough for simple memory-safe instruction sets such as CIL and the JVM.

To construct a CFG using the static control flow graph builder, we need classes from the following namespaces:

```
using Echo.ControlFlow.Construction;
using Echo.ControlFlow.Construction.Static;
```

### 1.7.1 The IStaticSuccessorResolver

The static graph builder of Echo defines one constructor that accepts an instruction set architecture, and a successor resolver. The architecture tells Echo how to interpret a model of an instruction in the instruction set. The successor resolver is able to tell the graph builder what the successors are of a single instruction. Both parameters are platform dependent, and every platform has their unique architecture representative and successor resolver.

In the following snippets, we assume our instruction model is called `DummyInstruction`, our architecture is represented by `DummyArchitecture`, and our successor resolver is of the type `DummyStaticSuccessorResolver`:

```
public readonly struct DummyInstruction { ... }
public class DummyArchitecture : IInstructionSetArchitecture<DummyInstruction> { ... }
public class DummyStaticSuccessorResolver : IStaticSuccessorResolver<DummyInstruction>
↪ { ... }

// ...

var architecture = mew DummyArchitecture();
var resolver = new DummyStaticSuccessorResolver(architecture);
```

### 1.7.2 Preparing the Instructions

Next we need a set of instructions to graph. We can use any instance of `IEnumerable<TInstruction>` to instantiate a `StaticFlowGraphBuilder`:

```
var instructions = new List<DummyInstruction> { ... }
var builder = new StaticFlowGraphBuilder<DummyInstruction>(architecture, instructions,
↪ resolver);
```

or use an instance of a class implementing the `IStaticInstructionProvider<TInstruction>` interface instead:

```
var instructions = new ListInstructionProvider<DummyInstruction>(architecture, new␣
↪List<DummyInstruction> { ... });
var builder = new StaticFlowGraphBuilder<DummyInstruction>(instructions, resolver);
```

### 1.7.3 Building the graph

Now we can build our control flow graph from our list. Given the entrypoint address stored in a variable *entrypointAddress* of type *long*, we can construct the control flow graph using:

```
ControlFlowGraph<DummyInstruction> graph = builder.
↪ConstructFlowGraph(entrypointAddress);
```

### 1.7.4 How it works

A static control flow graph builder performs a recursive traversal over all instructions, starting at a provided entrypoint, and adds for every branching opcode an edge in the control flow graph. By repeatedly using the provided `IStaticInstructionProvider` and the `IStaticSuccessorResolver` instances, it collects every instruction and determines the outgoing edges of each basic block.

The reason why a separate interface `IStaticInstructionProvider<TInstruction>` is used over a normal `IEnumerable<TInstruction>`, is because a normal list might not always be the most efficient data structure to obtain instructions at very specific offsets. Furthermore, this also allows for disassemblers to implement this interface, and decode instructions on-the-fly while simultanuously building the control flow graph.

This means it is a very efficient algorithm that scales linearly in the number of instructions to be processed, and it is usually enough for most simple instruction sets such as CIL from .NET or bytecode from the JVM, and could work for a lot of cases of native platforms such as x86.

### 1.7.5 Limitations

A big limitation of this approach, however, is that it cannot work on chunks of code that contain indirect jumps or calls. These might occur in for example chunks of x86 code such as the following:

```
mov eax, address
jmp eax
```

Since the static graph builder does not do any data flow analysis or emulation of the code, this basic block will produce a dead end in the final graph.

If this is a problem, dynamic graph builders (based on symbolic execution or emulation) might be more suited for the job, but might be significantly slower or expose the user to a risk of running arbitrary code on their own machine.

## 1.8 Symbolic CFG Construction

For some architectures, static recursive traversal of a procedure is not enough. Jump instructions might use the stack or registers to determine the branch target. For this, further analysis is needed and symbolic control flow graph builders could help.

To construct a CFG using the symbolic control flow graph builder, we need classes from the following namespace:

```
using Echo.ControlFlow.Construction;
using Echo.ControlFlow.Construction.Symbolic;
```

### 1.8.1 The IStateTransitionResolver

Similar to the static graph builder, the symbolic graph builder of Echo defines one constructor that accepts an instruction set architecture, and a transition resolver. The architecture tells Echo how to interpret a model of an instruction in the instruction set. The transition resolver is able to tell the graph builder how a certain instruction behaves and what its possible effects are, given a program state. Both parameters are platform dependent, and every platform has their unique architecture representative and transition resolver.

In the following snippets, we assume our instruction model is called `DummyInstruction`, our architecture is represented by `DummyArchitecture`, and our transition resolver is of the type `DummyTransitionResolver`:

```
public readonly struct DummyInstruction { ... }
public class DummyArchitecture : IInstructionSetArchitecture<DummyInstruction> { ... }
public class DummyTransitionResolver : IStateTransitionResolver<DummyInstruction> { ..
↪. }

// ...
```

(continues on next page)

(continued from previous page)

```
var architecture = mew DummyArchitecture();
var resolver = new DummyTransitionResolver(architecture);
```

### 1.8.2 Preparing the Instructions

Next, we need our instructions. We can use any instance of `IEnumerable<TInstruction>` to instantiate a `SymbolicFlowGraphBuilder`:

```
var instructions = new List<DummyInstruction> { ... }
var builder = new SymbolicFlowGraphBuilder<DummyInstruction>(architecture,␣
↪instructions, resolver);
```

or use an instance of a class implementing the `IStaticInstructionProvider<TInstruction>` interface:

```
var instructions = new ListInstructionProvider<DummyInstruction>(architecture, new␣
↪List<DummyInstruction> { ... });
var builder = new SymbolicFlowGraphBuilder<DummyInstruction>(instructions, resolver);
```

If decoding instructions requires more than just the current value of the program counter register, it is also possible to specify an `ISymbolicInstructionProvider<TInstruction>` instead. This takes an entire program state instead of just the program counter.

```
ISymbolicInstructionProvider<TInstruction> instructions = ...
var builder = new SymbolicFlowGraphBuilder<DummyInstruction>(instructions, resolver);
```

### 1.8.3 Building the graph

Building our control flow graph is then very similar to the static control flow graph builder. Given the entrypoint address stored in a variable `entrypointAddress` of type `long`, we can construct the control flow graph using:

```
ControlFlowGraph<DummyInstruction> cfg = builder.
↪ConstructFlowGraph(entrypointAddress);
```

A nice by-product of most symbolic transition resolvers is that it automatically also creates a data flow graph during the traversal of instructions.

```
DataFlowGraph<DummyInstruction> dfg = resolver.DataFlowGraph;
```

### 1.8.4 How it works

The symbolic graph builder traverses instructions in a similar way as a normal static graph builder would do. The difference is that while traversing, it maintains a symbolic state of the program, where it keeps track of the current state of the stack and variables as if the instructions were executed. Keep in mind though that the program state is fully symbolic, and does not actually execute the instructions.

This approach allows for transition resolvers to look at the current program state, and infer from this any indirect branch target or other unconventional behaviour.

For example, an x86 back-end could resolve the branch target of the following jump instruction, by looking at the data dependencies and recognising that the value of `eax` will contain the value *0x12345678* at runtime.

```
mov eax, 0x12345678
jmp eax
```

The downside is that because the symbolic graph builder needs to keep track of all the changes in a method body, it can be significantly slower and needs a lot more memory. Furtheremore, sometimes the builder might have to revisit a few instructions if more information has been obtained. Therefore, if no data flow graph is needed for the use case, it is recommended to use a static flow graph builder instead.

## 1.9 Control Flow Regions

Control flow graphs can be partitioned into one or more regions. This is used to encode certain relationships between nodes, such as modelling lexical scopes and exception handlers.

### 1.9.1 Types of regions

Echo defines three types of regions, which all implement the `IControlFlowRegion<TInstruction>` interface.

- `ControlFlowGraph`: The root region. Every node that is not part of any sub region is part of this region.
- `BasicControlFlowRegion`: A simple grouping of nodes and/or sub regions.
- `ExceptionHandlerRegion`: A region of nodes that is protected by one or more exception handlers, which are represented by other sub regions.

### 1.9.2 Interacting with regions

Nodes in a control flow graph are always part of a region. By default, they are part of the root region, which is the control flow graph itself.

Obtaining the region a node is situated in can be done using the `ParentRegion` property, or `GetSituatedRegions` method to get all the regions it is present in:

```
ControlFlowNode<TInstruction> node = ...

var directParentRegion = node.ParentRegion;
var allRegions = node.GetSituatedRegions();
```

Testing whether a node belongs to a region (including sub regions):

```
if (node.IsInRegion(parentRegion))
{
    // ...
}
```

Since it is often required to determine whether a node is inside of an exception handler regions, Echo defines a shortcut for finding the parent exception handler region:

```
ExceptionHandlerRegion<TInstruction> ehRegion = node.GetParentExceptionHandler();
if (ehRegion is null)
    Console.WriteLine("Node is not present in any exception handler.");
```

Moving a node to a region can be done using the `MoveToRegion`, or by adding it directly to the `Nodes` property of a `BasicControlFlowRegion` method.

```
BasicControlFlowRegion<TInstruction> otherRegion = ...;

node.MoveToRegion(otherRegion);
otherRegion.Nodes.Add(otherRegion);
```

Removing a node from any region can be done using the `RemoveFromAnyRegion` method (i.e. moving it back to the root region):

```
node.RemoveFromAnyRegion();
```

## 1.10 Exception handlers in CFGs

Exception handlers are special constructs that some platforms provide. Typically, in such a construct, one region of code (also known as the the try block in some languages) that is protected from exceptions or crashes. If any exception is thrown inside this region of code, control is transferred to one of the handler blocks that are associated to the protected region.

### 1.10.1 Exception Handler Models

Exception handlers are modelled using the `ExceptionHandlerRegion<TInstruction>` class, which implements `IControlFlowRegion<TInstruction>`, and can therefore be added to the *Regions* property of a control flow graph, or any other sub region inside the control flow graph.

The `ExceptionHandlerRegion<TInstruction>` class consists of two parts:

- The `ProtectedRegion`: This is also known as the try block of the exception handler. It is the sub region of the CFG that is protected by this exception handler from exceptions.
- The `HandlerRegions`: A collection of regions that represent the handler blocks. Control might be transferred to one of these regions whenever an exception occures in the protected region.

### 1.10.2 Detecting Exception Handler Regions

Echo can automatically create and subdivide a given CFG into exception handler regions if it is provided a list of address ranges representing the exception handlers.

```
ControlFlowGraph<TInstruction> cfg = ...

// Define exception handler ranges.
var ranges = new[]
{
    new ExceptionHandlerRange(
        protectedRange: new AddressRange(tryStartOffset, tryEndOffset),
        handlerRange: new AddressRange(handlerStartOffset, handlerEndOffset)),
    ...
};

// Subdivivde the control flow graph into exception handler regions.
cfg.DetectExceptionHandlerRegions(ranges);
```

If two `ExceptionHandlerRange` s have the same address range for their protected region, they will be merged into one `ExceptionHandlerRegion<TInstruction>`, and both handler regions will be added to the same `ExceptionHandlerRegion<TInstruction>` region.

Exception handler ranges can also be assigned some user data. This allows for associating these ranges to additional metadata like the exception type that is caught:

```
object someMetadata = ...

var range = new ExceptionHandlerRange(
    protectedRange: new AddressRange(tryStartOffset, tryEndOffset),
    handlerRange: new AddressRange(handlerStartOffset, handlerEndOffset),
    tag: someMetadata),
```

This additional metadata is then added to every handler region's `Tag` property.

### 1.10.3 Common problems with detecting exception handlers

**I cannot find any exception handler regions in the CFG after calling DetectExceptionHandlerRegions:**

The *DetectExceptionHandlers* method requires that all nodes are present in the graph to function properly. Make sure that this also includes all nodes from any handler block.

On some platforms that implement exception handlers, the handler blocks are considered unreachable through normal execution paths. Therefore, if the control flow graph is built using a static or symbolic control flow graph builder, these nodes are not reached (as they perform a recursive traversal). To solve this issue, you can provide these flow graph builders these handler blocks as known basic block headers:

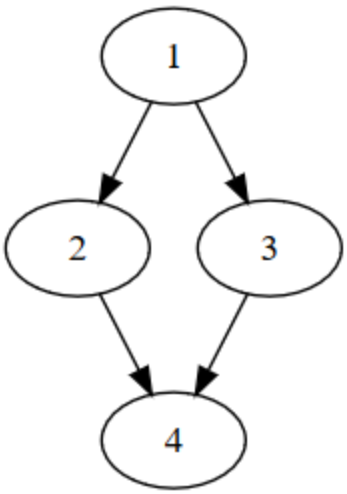
```
IFlowGraphBuilder<TInstruction> builder = ...

var cfg = builder.ConstructFlowGraph(instructions, entrypointAddress, ranges);
```

## 1.11 Dominator Analysis

### 1.11.1 What are dominators?

Suppose we have two nodes A and B, residing in a control flow graph (CFG). We say that node A dominates node B if and only if for executing node B, the program has to go through node A. A node always dominates itself.

In the example below, node 1 dominates all nodes in the graph. However, node 2 does not dominate node 4, since there exists another execution path to node 4 that does not include node 2 (namely 1 -> 3 -> 4).



Dominator analysis give structure to CFGs, and can be very useful in static analysis to detect various constructs in a CFG such as loops, and reveal the importance of various nodes in terms of control flow.

### 1.11.2 Dominator analysis in Echo

To perform dominator analysis on a CFG produced by Echo, we can construct a dominator tree by the following:

```
using Echo.ControlFlow.Analysis.Domination;
...
var dominatorTree = DominatorTree.FromGraph(graph);
```

A *DominatorTree* has a `Root` node, and every node in the tree corresponds to one node in the original control flow graph. Furthermore, a dominator tree can be used to verify whether a specific node dominates another. Taking the example in the figure above, we have that:

```
dominatorTree.Dominates(n1, n4) // returns true
dominatorTree.Dominates(n2, n4) // returns false
```

The *DominatorTree* class implements the `IGraph` interface, which means it can also be used with various other analysis algorithms (such as traversals) and export utilities (such as exporting to dot files).

## 1.12 Blocks

An important feature of Echo's control flow analysis package is the representation of code blocks. Blocks describe the structural hierarchy and context of nodes in a control flow graph, and can be used to get more insight about which nodes belong to each other, and in which order blocks appear. They also form an integral part in the serialization from control flow graphs back to a raw sequence of instructions.

### 1.12.1 Types of blocks

Echo defines three types of blocks that might appear in a block tree, which all implement the `IBlock<TInstruction>` interface:

- `BasicBlock`: Represents a sequence of instructions that when executed, is executed in its entirety, and can only have incoming branches at the very beginning of the block, and can only introduce outgoing branches by the last instruction in the block.
- `ScopeBlock`: Represents a sequence of blocks that are put into a lexical scope. A lexical scope block can contain any other types of blocks, including nested scope blocks.
- `ExceptionHandlerBlock`: Represents a scope block that is protected by one or more exception handlers.

The entirety of a function's body is usually represented using a single `ScopeBlock`. To get all basic blocks (leafs in the blocks tree), use the `GetAllBlocks` method. This can for example be used to get all instructions in a block tree.

```
IBlock<TInstruction> someBlock = ...;

foreach (BasicBlock<TInstruction> basicBlock in someBlock.GetAllBlocks())
{
    Console.WriteLine($"Block_{basicBlock.Offset:X}:);

    foreach (var instruction in basicBlock.Instructions)
        Console.WriteLine(instruction);
```

(continues on next page)

(continued from previous page)

```
    Console.WriteLine();
}
```

### 1.12.2 Constructing Blocks from Control Flow Graphs

Constructing structural blocks can be done by using the `BlockBuilder` class, defined in the `Echo.ControlFlow.Serialization.Blocks` interface. Below an example on how to construct a `ControlFlowGraph<TInstruction>` into an instance of `ScopeBlock<TInstruction>`:

```
ControlFlowGraph<TInstruction> cfg = ...;

BlockBuilder<TInstruction> builder = new BlockBuilder<TInstruction>();
ScopeBlock<TInstruction> rootScope = builder.ConstructBlocks(cfg);
```

**Warning:** While the block builder sorts blocks in a control flow graph in a topological sorting, it does not introduce any branch instructions that might be needed to correctly connect the blocks.

### 1.12.3 Block visitors and listeners

The blocks API implements the visitor pattern. That is, once a `ScopeBlock<TInstruction>` is constructed, it can be traversed using the `IBlockVisitor<TInstruction>` interface, which defines methods for each type of block that might appear in a blocks tree. Every type of block defines the `AcceptVisitor` method, that calls the corresponding visitor method in the `IBlockVisitor<TInstruction>` interface.

Since a pre-order traversal is one of the most common traversals of a block tree, this is implemented using the `BlockWalker<TInstruction>` class. To get ahold of all the blocks that are entered and exit, the block walker takes any instance of `IBlockListener<TInstruction>` that defines `Enter` and `Exit` methods for every type of block. An example implementation is the `BlockFormatter` class, which converts every encoutered block to a human-readable string.

```
IBlock<TInstruction> someBlock = ...;

var formatter = new BlockFormatter<TInstruction>();
var walker = new BlockWalker(formatter);
walker.Walk(someBlock);

string output = formatter.GetOutput();
Console.WriteLine(output);
```

## 1.13 The Basics

### 1.13.1 What are Data Flow Graphs?

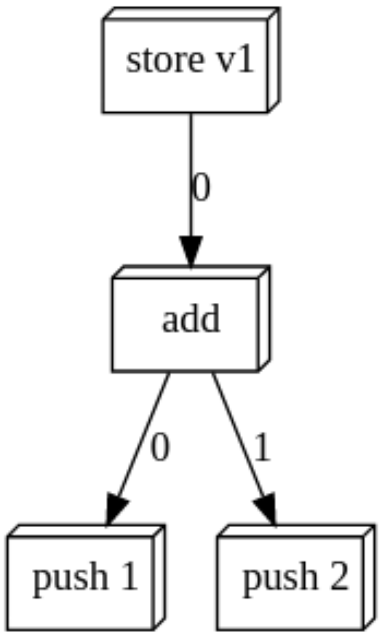
Data flow graphs are graphs that encode the dependencies between objects, instructions or components. They can be used to do dependency analysis, or

Consider the following excerpt of a function written in a fictional stack-based language:

```
push 1
push 2
add
store v1
```

From the example above, we can see that two constants are pushed onto the stack, then an addition is applied, and the result is stored into `v1`.

A data flow graph would model this as the following:




In the graph, every outgoing edge node encodes a data dependency to another node.

### 1.13.2 Data Flow Graph Models

There are three main classes that are used to model DFGs. Given a `TContents` type, representing the type of data to store in each node of the DFG, Echo defines the following classes:

- `DataFlowNode<TContents>`: A single node, wrapping an instruction or component.
- `DataDependency<TContents>`: A stack or variable value that an instruction or component depends on, that keeps track of a collection of data sources.
- `DataSource<TContents>`: A single data source referenced by a `DataDependency<TContents>`. This object translates to a single edge in the data flow graph.
- `DataFlowGraph<TContents>`: A collection of nodes and their dependencies.

These classes implement the `INode`, `IEdge` and `IGraph` interfaces, and work therefore with all kinds of generic graph algorithms and export features.

### 1.13.3 Data flow nodes

Nodes expose their direct dependencies using the following properties:

- `StackDependencies`: A collection of stack slots that the node depends on.

- `VariableDependencies`: A collection of variables the node depends on.

These properties are collections of type `DataDependency<TContents>` and are fully mutable, and can be used to add or remove dependencies between nodes. Every `DataDependency<TContents>` represents a single value with potentially multiple data sources. As a result, every stack value that an instruction pops is assigned exactly one `DataDependency<TContents>` in the `StackDependencies` property, and every value that was read from a variable is put in the `VariableDependencies` property.

### 1.13.4 Simple data dependencies

Taking the example on the top of this article, the `store v1` instruction would have one single data dependency with one data source to the `add` instruction, and the `add` instruction would have two data dependencies with both one data source to the `push 1` and `push 2` instructions respectively.

Below an example snippet that obtains the dependency instructions of the `add` instruction.

```
DataFlowGraph<T> dfg = ...
DataFlowNode<T> add = dfg.GetNodeById(...);

// Obtain the symbolic values that are popped by the instruction.
DataDependency<T> argument1 = add.StackDependencies[0];
DataDependency<T> argument2 = add.StackDependencies[1];

// Get the instructions responsible for pushing the values
DataFlowNode<T> argument1Source = dependency1.First().Node; // returns the node␣
↪representing "push 1"
DataFlowNode<T> argument2Source = dependency2.First().Node; // returns the node␣
↪representing "push 2"
```

To speed up the process, Echo defines an extension methods that obtains all of the dependency data flow nodes, and sorts them in such a way that they can be evaluated in order.

```
DataFlowGraph<T> dfg = ...
DataFlowNode<T> add = dfg.GetNodeById(...);

var dependencies = add.GetOrderedDependencies(); // returns {"push 1", "push 2"}
```

By default, `GetOrderedDependencies` traverses all edges in the data flow graph. This includes variable dependencies that were registered in the graph. If only the stack dependnecies are meant to be traversed (e.g. to get the instructions that make up a single expression), additional flags can be specified to alter the behaviour of the traversal.

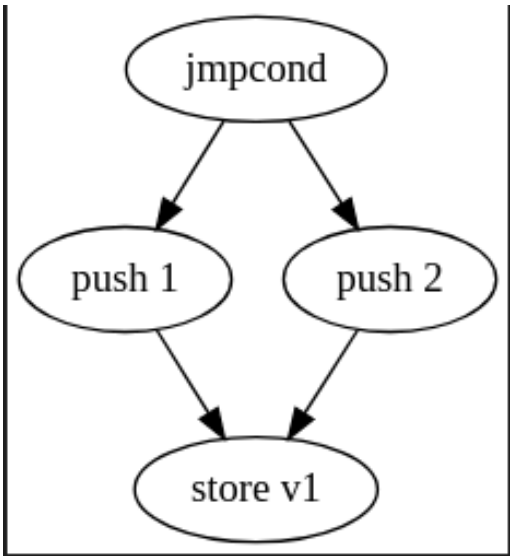
```
DataFlowGraph<T> dfg = ...
DataFlowNode<T> add = dfg.GetNodeById(...);

var dependencies = add.GetOrderedDependencies(DependencyCollectionFlags.
↪IncludeStackDependencies);
```

**Warning:** When a data dependency has multiple data sources, `GetOrderedDependencies` will only choose one. The method is defined to find one sequence of instructions that produce the values of the dependencies, not all possible sequences of instructions. It is undefined which sequence is picked.

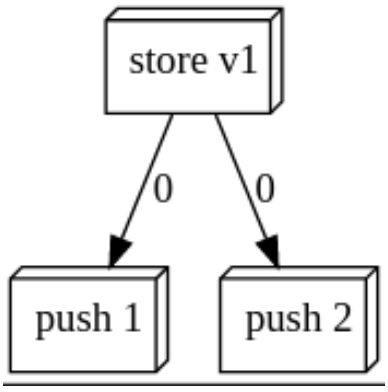
### 1.13.5 Multiple data sources

It is important to note that a single data dependency might have multiple data sources (i.e. where the value might come from).

For example, if we change the example slightly to the following control flow graph:




The value that is popped by the `store v1` instruction has two possible values, and therefore two different data sources. In Echo, this would be encoded as a single `DataDependency<TContents>` with two possible data sources `push 1` and `push 2`. The resulting data flow graph would therefore look something like the picture below:




Below an example on how to find the direct dependencies of the `store v1` node:

```
DataFlowGraph<T> dfg = ...
DataFlowNode<T> storeV1 = dfg.GetNodeById(...);

// Obtain the symbolic value that is popped by the instruction.
var dependency = storeV1.StackDependencies[0];

// Print out the possible data sources for this value:
foreach (DataSource<T> source in dependency)
    Console.WriteLine(sourceNode.Node.Contents);
```

## 1.14 DFG Construction

Constructing data flow graphs from a list of instructions can be done by constructing a control flow graph using the `SymbolicFlowGraphBuilder`. This class generates both a control flow graph, as well as a data flow graph as a by-product. See the section `Symbolic CFG Construction` for more details.