University of Twente
Faculty of Electrical Engineering, Mathematics & Computer Science
Formal Methods and Tools

Graph-Based Semantics of the .NET Intermediate Language

by

N.B.H. Sombekke

May, 2007

Graduation Committee: dr. ir. A. Rensink (1st supervisor)
ir. H. Kastenberg
ir. T. Staijen

Abstract

The semantics of a programming language are often described in a natural language. Such descriptions are often ambiguous and hard (or even impossible) to construct in a precise way. To tackle these problems one could specify a formal description of the semantics by using a mathematical model. In this report such a mathematical model is presented for the .NET Intermediate Language (IL) in the form of graphs and transformations to these graphs.

In order to be able to perform transformations on graphs, we need a start graph. The .NET Intermediate Language generates bytecode and cannot supply such a start graph. Therefore we have constructed a translator that translates an arbitrary IL program into a so called Abstract Syntax Graph (ASG). The ASG is the start graph to which we now can apply graph transformations.

Central in this report are graph production rules that we have specified in order to describe the semantics of .NET IL instructions. These production rules are used for transforming graphs, and by applying them to the (intermediate) graphs repeatedly it is possible to simulate a program.

Although further research is necessary, we believe this project provides a promising basis of representing the semantics of the .NET Intermediate Language in an intuitive and formal way.

Contents

Chapter 1

Introduction

Probably everybody has experienced a time that a communication problem between two persons appeared. For example, when your mother asked you to get a bread from the bakery and that when you were at the bakery you did not know what bread to take.

Formally speaking, executing the task can have a different result than the person who gave the task had in mind. Something similar holds for the meaning and behaviour (semantics) of programming languages. When describing the semantics of programming languages in a natural language (such as English), this can lead to ambiguity. Furthermore, by using a natural language to describe semantics of a programming language it is hard (or even impossible) to present the semantics in a precise way. It is also easy to introduce mistakes or forget details. Take for example an instruction which adds two values. It is easy to forget specific details of where these two values can be found, or where the result of this operation should be stored.

To tackle these problems, one should specify the semantics in a formal way. This specification is represented in the form of a mathematical model and can serve as a basis for understanding and formal reasoning about the behaviour of programs. It is also useful for precisely defining the meaning of program constructions. When giving a formal description, details that are normally easily overseen will be unrevealed and made explicit, leaving no space for ambiguity. Another advantage of using a mathematical model is that it opens possibilities for analysis and verification.

There are several formal languages for expressing semantics of a programming language. The *Structural Operational Semantics* (SOS) approach, introduced by Plotkin[20] in 1981, has been very popular. SOS generates a transition system by using logical rules. Due to these logical rules, SOS can be hard to grasp for persons unfamiliar with logic. Also see [1, 27] for more information about SOS.

A more recent technique of giving a formal description of semantics is by using *graphs* and applying *transformations* to these graphs. A graph is used to model a state of a program, and the graph transformations are an intuitive, easy to understand, and a clear way to express the behaviour of the program in a rule-based way, just as with SOS. Using graphs is especially useful for representing object-oriented programs because the states of these programs mainly depend on a set of reference values. Furthermore, graph transformations lend themselves for describing dynamic changes to such states. To be able to work with graphs and graph transformations, a tool is needed to construct graphs and perform transformations to these graphs. There exist quite some tools on this, but in this work we use the GROOVE tool set[9].

The *Graphs for Object-Oriented Verification* (GROOVE) project aims at using graphs to model the structure of object-oriented programs and using graph transformations to model operational semantics[9]. As part of the GROOVE project, a tool set has been developed. Among others, this tool set consists of an editor that can be used to construct graphs and graph transformation rules (which we also call *production rules*). The tool set also contains a simulator that is able to apply production rules to graphs. Each time a production rule is applied, a new graph is constructed. When we use graphs as a representation of states, and the application of production rules as transition between these states, we can uses these graphs and production rules to construct a

transition system. In such a transition system states are represented by nodes, and transitions between states are represented by edges.

This thesis shows that graph transformations can be used for the specification of the semantics of a programming language, in particular the *.NET Intermediate Language*. We have chosen the .NET Intermediate Language because this is a low-level intermediary language which covers all other .NET languages. The relation of the .NET Intermediate Language and the other .NET languages can be explained using the analogy of an interpreter that translates Russian and Dutch to English. Here, English is analogous to the Intermediate Language, and Russian and Dutch to the other .NET languages. If you are able to understand English, you will also be able to understand the other languages when talking via the interpreter.

1.1 Problem Statement

The semantics of programming languages described by using natural languages can be ambiguous, meaning that a text can easily be open for multiple interpretations. On the other hand, pictures in the form of graphs and graph transformations are often more clear and less ambiguous than natural languages. Also, it is possible to reason about graphs and graph transformations thanks to their formal background [25]. Another motivation to use a graph-based representation is that graphs are a convenient way to represent the structure of a program, and graph transformations are a good technique to represent object-oriented behaviour.

Our interest is that we want to describe the semantics of an object-oriented programming language using graph transformations. We have chosen for the .NET Intermediate Language (IL) because all .NET languages are compiled to IL, which makes it attractive to do our research on IL instead of other .NET languages individually.

The main question now is: how can we describe the semantics of the .NET IL by using graph transformations? Therefore, the goal of this research project is the development of a graph-based specification of the semantics of the Microsoft .NET IL.

1.2 Approach

To accomplish the graph-based representation, we decided to construct a set of graph production rules specifying the semantics of the IL. When modelling the syntax and states of IL programs as graphs accordingly, we can apply the rules to those graphs, i. e. simulate the program. Figure 1.1 contains an overview of these two steps.

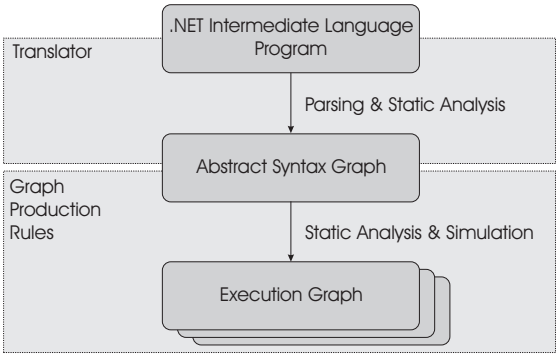


Figure 1.1: From program to simulations, an overview.

This figure shows that a translator translates a .NET Intermediate Language program into the Abstract Syntax Graph, to which graph production rules are applied. Both the *translator* and the *graph production rules* do not exist yet and need to be constructed by us. The following paragraphs provide a description of these components.

Translator We need a translator to construct a graph from an arbitrary IL program. This graph is an abstract syntax representation of the IL program, and therefore is called an Abstract Syntax Graph (ASG). The ASG contains the structure of the program, the instructions that have to be executed and the syntactic order of these instructions. During the static analysis phase, we enrich the ASG by creating and adding method signatures, transforming namespaces, and resolving identifiers.

Graph Production Rules Although static analysis is mostly performed in the translator, a minor part of static analysis will be performed by using a graph production rule in order to provide some intuition of what happens during static analysis. This involves merging of labels having the same identifier. Furthermore, the ASG contains implicit control flow information. A decision has to be made whether or not to make this control flow explicit by using a control flow analysis phase.

The major goal of this project is the specification of the semantics of the .NET Intermediate Language by graph transformations. We aim at specifying the semantics with one or two transformation rules per instruction. These graph production rules transform a graph, which in fact is the Abstract Syntax Graph delivered by the translator, into another graph. Each time a production rule from this graph transformation system is applied, a so called Execution Graph is created. Such a graph is a representation of a system state. Thus by applying the whole graph transformation system to the Abstract Syntax Graph, we can *simulate* the IL program. By simulating the program a *transition system* is constructed, which consists of Execution Graphs (states) and the applied production rules (transitions) to these Execution Graphs .

1.3 Overview

This thesis is organized as follows: First, some background information will be provided. Therefore, Chapter 2 presents an overview of the .NET framework and its most important parts; among others, the common language runtime, common type system, types and garbage collection. Chapter 2 also contains a brief introduction into the .NET Intermediate Language and its instructions. Graph transformations and their use are described in Chapter 3. In this chapter we will also tell something about the GROOVE tool set. Chapter 4 discusses the implementation of the translator and some problems that are encountered. The specification of the graph transformation rules, which represent the semantics of the Intermediate Language, is elaborated in Chapter 5. This chapter also includes an explanation of the encountered design problems and their solutions. Finally, a discussion about the project and the conclusions are presented in Chapter 6.

Chapter 2

The .NET Framework

In the year 2000 Microsoft launched the .NET (pronounced: *dot net*) initiative. The .NET Framework is a development and runtime infrastructure that can be used for the development of applications for the Windows platform.

The framework is designed to fulfil the following objectives [16]:

- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of interpreted or scripted environments.
- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.
- To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

This chapter contains an introduction to the .NET Framework and the Intermediate Language. First we will present an overview of the .NET Framework, what it contains and how it works. After that, more will be explained about the Intermediate Language. Most figures presented in this chapter are derived or based on figures from [19] and [6].

2.1 Overview of .NET

A .NET program is written in a programming language that uses the .NET runtime as its execution environment. That is, the sources of this program are compiled, using a language compiler for the specific programming language, to an intermediate format called the Common Intermediate Language (CIL) or just Intermediate Language (IL). The intermediate format is accepted by the Common Language Runtime which uses just-in-time (JIT) compilation to compile the intermediate format to CPU specific code (also called native machine code). After this, the CPU specific code can be executed. A schematic overview of this principle is given in Figure 2.1.

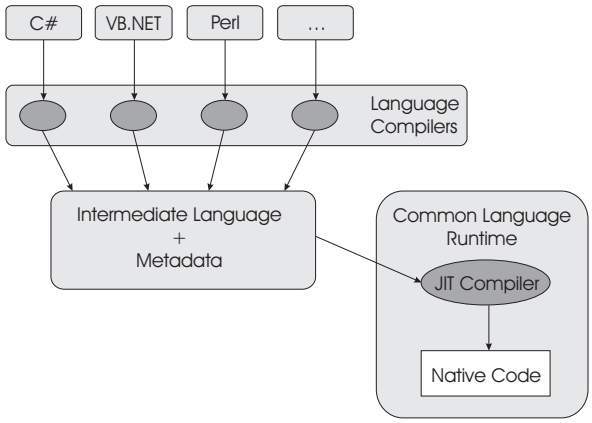


Figure 2.1: Overview of Languages, Intermediate Language and Common Language Runtime

2.1.1 Common Language Runtime

The Common Language Runtime (CLR) is the runtime environment in which .NET applications are executed. Consider the CLR to be comparable to the Java Virtual Machine (JVM) (see [11]). Like JVM uses an intermediate language representation of Java (called bytecode), the CLR uses IL. IL code is sometimes referred to as *managed code* because the CLR manages its lifetime and execution [24]. To do this, the CLR provides necessary core services such as memory and thread management and strict type safety. Code that does not target the runtime is known as *unmanaged code*. Unmanaged code is for example native code (i. e. machine code).

The CLR uses a just-in-time (JIT) compiler to compile the IL code, which is stored in a so called *Portable Executable* file, into native (platform-specific) code. After this conversion, the native code is executed. This means that .NET code is always *compiled*, not interpreted. The usage of IL code and JIT compilation ensures that code is portable as well as efficient.

2.1.2 Base Class Library

The Base Class Library (BCL), sometimes also referred as .NET Framework Class Library (FCL) [24], is a library containing all important types (i. e. classes) of the .NET Framework. The BCL is language independent, meaning that it does not depend on the used .NET language. Furthermore, the BCL is available for all languages using the .NET Framework. The class library encapsulates a number of common functions such as file reading and writing, network programming and graphic rendering.

2.1.3 Common Type System and Common Language Specification

When different languages must cooperate with each other, some kind of agreement must exist on how this is accomplished. Therefore, the *Common Type System* (CTS) is defined within the CLR, making it a fundamental part of the CLR. It defines the entire set of types that can be used with many different language syntaxes and makes it possible for two different .NET languages to use each other's objects. Language compilers targeting the CLR must generate code that is conformant to the CTS. The CTS performs the following functions [16]:

- Establish a framework that helps enable cross-language integration, type safety, and high performance code execution.
- Provide an object-oriented model that supports the complete implementation of many programming languages.

- Define rules that languages must follow, which helps ensure that objects written in different languages can interact with each other.

It is possible for one language to allow a construct supported by the CTS, while another language does not. This can be a barrier for cross-language integration. Therefore, the *Common Language Specification* (CLS) is developed, which is a subset of the CTS. The CLS is a set of of basic language features needed by many languages[17]. It includes language constructs often required by many software developers, but is small enough for most languages to be able to support it.

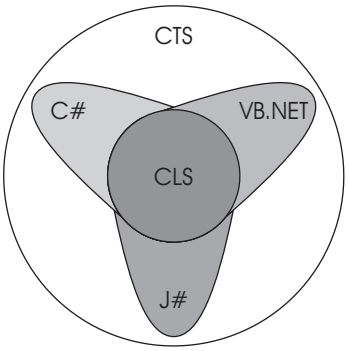


Figure 2.2: Common Type System and Common Language Specification

For more information about the CTS and the CLS, see [6].

2.1.4 Types

The CLR distinguishes between *value types* and *reference types* [6, 19, 26].

Value types are used to describe values. Values are instances of value types. They are directly stored at the memory address on the method stack assigned by their variable, or inside an object in case of a field of an object. Value types must always contain some data and thus cannot be null. When passing value types as argument to a function, a copy of the value is made prior to function execution. Thus, when executing the function, the copy of the value is used and can be changed, but the original value persists.

Reference types contain references to heap-based objects and can be null. Reference types include classes, interfaces and arrays. When reference types are passed as an argument to a function, the pointer to the object is passed (unlike in case of the value types where a copy of the object is passed). Thus, passing by reference means that changes will be made to the original object.

In Figure 2.3 a diagram containing the different value and reference types is presented. Here it is clear what the predefined value and reference types are and what kind of user-defined types can be created. What we omitted up to now is that it is also possible to create a reference type of a value type by a technique called boxing. For more information about boxing, we refer to [6].

The list of predefined types is shown in Table 2.1. The table contains a description of the type, a mapping to the .NET class library and whether or not the type is supported by the CLS.

2.1.5 Portable Executables

Compiling a .NET program results in one or more files containing IL code and metadata. .NET programs are stored in a binary format that is compatible with the Windows binary format PE

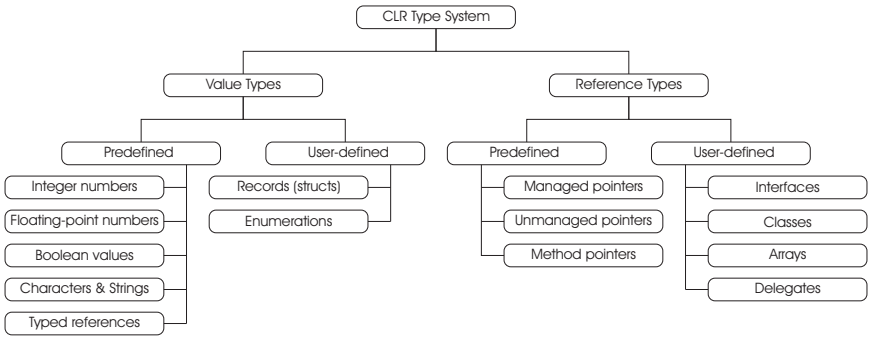


Figure 2.3: Types supported by the CLR

Table 2.1: Predefined Types

CIL Name	**CLS Type?**	**Name in BCL**	**Description**
bool	✓	System.Boolean	1 byte: 0 (= false), 1-255 (= true)
char	✓	System.Char	16-bit Unicode character
string	✓	System.String	Unicode character string
float32	✓	System.Single	IEEE 32-bit floating-point number
float64	✓	System.Double	IEEE 16-bit floating-point number
int8		System.SByte	Signed 8-bit integer
int16	✓	System.Int16	Signed 16-bit integer
int32	✓	System.Int32	Signed 32-bit integer
int64	✓	System.Int64	Signed 64-bit integer
unsigned int8	✓	System.Byte	Unsigned 8-bit integer
unsigned int16		System.UInt16	Unsigned 16-bit integer
unsigned int32		System.UInt32	Unsigned 32-bit integer
unsigned int64		System.UInt64	Unsigned 64-bit integer
native int	✓	System.IntPtr	Machine-dependent signed integer number (2's-complement)
native unsigned int	✓	System.UIntPtr	Machine-dependent unsigned integer number
object	✓	System.Object	Managed pointer to object on heap
typedref		System. TypedReference	Pointer plus exact type

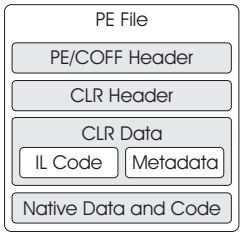


Figure 2.4: Portable Execution File

(Portable Executable). A PE file is not executable by itself, it is the CLR that compiles PE files into native code. The layout of a typical .NET PE file is shown in Figure 2.4.

The sections have the following meanings:

- The *PE/COFF header* is loaded by the operating system. It indicates the type of file:

Graphical User Interface (GUI), Character-based User Interface (CUI) or Dynamic-Link Library (DLL). One of the components stored in the header is a timestamp indicating when the file was built. Furthermore, the PE/COFF header contains references to other contents within the PE file. For modules targeting the CLR there is information available allowing the runtime to seize control.

- The *CLR header* indicates that the PE file is a .NET executable. The most important components of the CLR header are the required version of the CLR, some flags, and possibly a description of the entry point method of the executable. The runtime header, which contains all of the runtime-specific data entries and other information, should reside in a read-only, shareable section of the image file.

- The *CLR data* section contains *metadata* and IL code. The metadata section contains two parts: tables that describe the types and members defined in the source code, and tables that describe the types and members referenced by the source code. The *Intermediate Language code* is created by the compiler that compiled the source language. This IL code will eventually be compiled into native machine code by the CLR.

- The *Native Data and Code* section contains native code, for example precompiled C++ to machine code.

Although the PE file contains different sections, we are only interested in the IL Code section. The IL Code section contains the IL program that is eventually simulated.

2.1.6 Virtual Execution System

In the CLR, program execution is performed by a number of components interoperating under the name Virtual Execution System (VES). The VES is also known as the Execution Engine. An overview of the VES is presented in Figure 2.5. The VES is, among other things, responsible for loading a PE file (containing the IL program), the translation from IL into machine code, and for its execution.

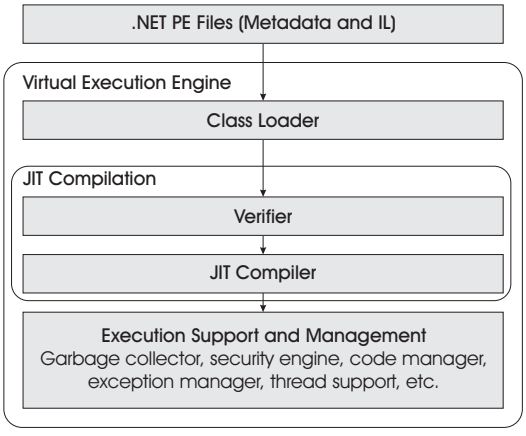


Figure 2.5: Overview of the Virtual Execution System (VES)

This thesis is restricted to the simulation of execution by the code manager. The rest of the VES falls outside the scope of this project.

2.1.7 Code Management

Stack and Heap

A *stack* is a data structure that works according the *Last In First Out* (LIFO) principle. Values can be respectively put on (*pushed*) and pulled from the top of the stack (*popped*). It is not possible to store or retrieve values of the stack, other than the top value. Additionally it is not possible to just read the top-of-stack value, without pulling it from the stack.

The Common Language Runtime is stack-based. This means that the CLR uses a stack to store intermediate values on. This stack is not addressable by other methods and is initially empty on each method call. On leaving a method, the stack only contains a return value (if available).

The *heap* is a dynamic storage area in which objects of classes and arrays can be stored. References to objects in the heap are stored by means of pointers on the stack. It is also possible that objects in the heap contain references to other objects.

An instance of a value type has its value stored on the stack (or in a containing object in the heap), meaning that a piece of memory is reserved for their value. Instances of reference types have a reference to heap-based objects allocated on the stack. Instances of reference types can be null.

Memory Management

When a method is called, a *method state* (which contains the information captured in an invocation stack frame) is created. A method state contains all information about the environment within which a method executes. It contains an instruction pointer that points to the next IL instruction to be executed within the current method. It also contains an evaluation stack that is entirely local to the method, and thus cannot be accessed by other methods. The contents of the evaluation stack are preserved across call-instructions.

Both *Input parameters* (i. e. the arguments of the method) and local variables are stored in ordered lists that are addressable via an index. The values of both input parameters and local variables are preserved across method calls.

The *local memory pool* is used for dynamic allocation of storage space which is not freed by the garbage collector. The storage space will be reclaimed on method exit [19]. The local memory pool is used to allocate objects which type or size is not known at compile time and which the programmer does not wish to allocate in the managed heap [6].

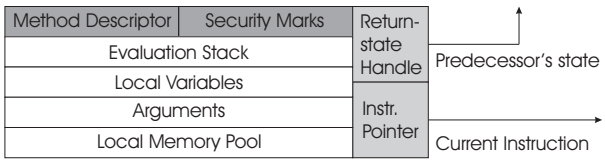


Figure 2.6: Method state

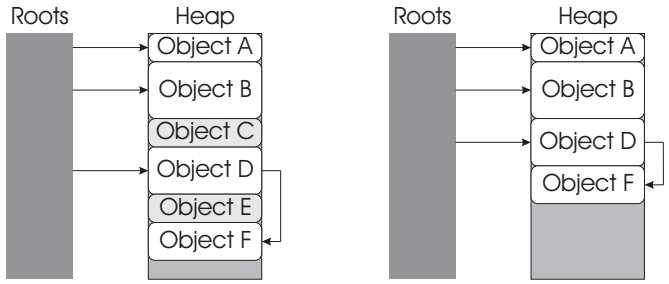
When a method is called, the new method state is appended to the end of a list of method states (method or procedure stack) and linked to its predecessor. The caller method is stored in the *return-state handle*. When returning to the caller method, the results of the method that is exited must be copied to the stack of the caller method and the method state must be removed from the list of method states because it is no longer needed. The return-state handle is used to restore the method state on return from the current method [6]. This corresponds with the *dynamic link* compiler terminology.

2.1.8 Garbage Collection

Garbage collection is the process of identifying and cleaning up unused data in the managed heap to reclaim memory. The garbage collector automatically detects and removes objects that are not longer referenced.

By default, garbage collection takes place when the system runs out of memory and no space is available to create new objects. At such a moment the garbage collector starts running. The garbage collector is responsible for suspending all active threads and marking all objects in the heap as garbage. After that, the garbage collector builds a graph for all objects reachable from the roots of the program. The roots of the program identify storage locations that refer to objects on the heap or to objects that are set to null. Once all roots have been checked (the graph contains only the objects reachable from the program's roots), the objects not contained in the graph are considered garbage. The memory space used by these objects now can be freed and non-garbage objects are shifted down in memory to remove gaps in the heap. Because objects now are positioned on other memory addresses, the pointers to these objects now become invalid and must be updated by the garbage collector with the new memory address. Once these pointers are updated, the suspended threads can be restarted and the garbage collection phase is finished.

In Figure 2.7 an example of the heap before and after garbage collection is presented. The heap prior to garbage collection (Figure 2.7(a)) contains unreferenced objects (Object C and Object E), which are deleted during garbage collection resulting in the heap displayed in Figure 2.7(b).



(a) Before (b) After

Figure 2.7: Example representation of the heap before and after garbage collection

The advantage of garbage collection is that objects are cleaned up automatically and thus do not have to be tidied up manually (which causes memory leaks when forgotten). The main disadvantage of garbage collection is that it introduces a performance hit and also that the execution of all active threads must be suspended in order to apply garbage collection.

For more information about garbage collection we refer to [23, 19].

2.2 The Intermediate Language

The .NET Framework uses language compilers that target the Common Language Runtime. For instance, Microsoft provides C#, J#, VB .Net, Jscript .Net, and C++ compilers. Furthermore, there are third-party compilers that target the CLR, such as an Eiffel, Cobol or Perl compiler.

As mentioned before, the source code of a .NET supported language is compiled to an intermediate format called the Intermediate Language (IL). The IL includes instructions for loading, storing, initializing and calling methods on objects, as well as instructions for arithmetic and logical operations, control flow, direct memory access, exception handling and other operations [16].

Together with the produced IL, metadata is generated. Metadata contains, among other data, a description of the types in the code, their members, and code references. The IL and metadata are contained in a so called *assembly*. The assembly is constructed using the portable executable (PE) file format that is based on and extends the published Microsoft PE and common object file format (COFF) used for executable content. The PE filetype accommodates IL, native code and metadata. The presence of metadata in the file, along with the IL, enables written code to describe itself. The runtime locates and extracts the required metadata from the file during execution [16, 18, 15].

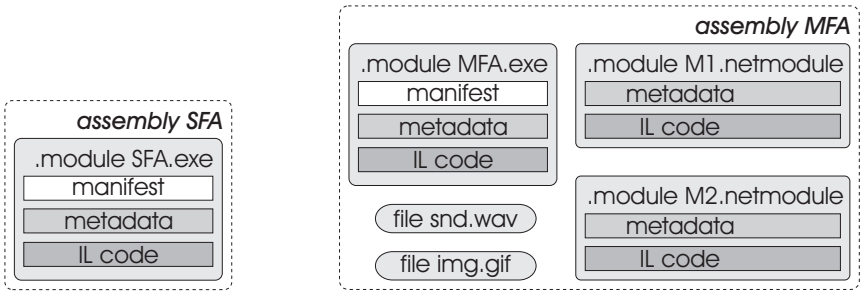
2.2.1 Directives

Directives are bits of metadata representing the components which compose our program. They are are not actual IL instructions representing code [5]. Directives can ask the runtime-environment to perform some task and can be recognized in IL as productions starting with a period (`.`). For example, a method containing directive `.maxstack` n means that at most n stack slots are required. For a complete list of directives we refer to the CLI Specification [6].

2.2.2 Modules and Assemblies

Modules are single files (PE-files, see Section 2.1.5) that contain executable code targeting the Virtual Execution System (VES). As stated above, a module contains type definitions and IL code.

One or more modules can be embedded in an assembly. An assembly is a logical unit of functionality, containing one or more modules. Thus, a .NET application can be packaged into assemblies, which respectively are called a single-file assembly and a multi-file assembly. The latter can also contain resources as images or sounds. In Figure 2.8 both a single-file assembly and a multi-file assembly are represented.



(a) Single-file assembly (b) Multi-file assembly

Figure 2.8: Difference between a single and multi-file assemblies and its modules

Note that the multi-file assembly contains multiple modules (which are physical files) and that one of these modules contains a *manifest*. The manifest contains information for finding all the definitions of an assembly, which is important for loading and running the other modules within the assembly. Also note that the multi-file assembly can contain images (file `img.gif`) and sounds (file `snd.wav`).

To simplify this research project we only use single-file assemblies.

2.2.3 Namespaces

The namespace concept is used to group functionality within unique names. The name of the namespace is often the same as the name of the file in which the code exists, but it is also possible to have multiple namespaces in one single file or to have a namespace that spans over multiple files. To prevent equally named namespaces colliding with each other, they are contained within assemblies.

The Intermediate Language has no distinct concept of *current namespace*. A type is always referred to by its full name, relative to the assembly in which it is defined.

2.2.4 Methods

Operations associated with a type or with instances of a type are called methods. There are two types of methods, namely static methods (class methods) and instance methods. The major difference is that static methods are not connected to an object and cannot access any object

methods or attributes. A static method thus is only associated with the type itself, instead of with an instance of that type. Static methods do not have an instance pointer (*this*). The arguments of static methods are indexed, starting with 0.

Instance methods are methods that are associated with an instance of a reference type, and can be virtual and nonvirtual. Virtual methods are those that can be replaced and overridden by subclasses, whereas nonvirtual methods cannot. Instance methods have access to the *this* pointer as unlisted first argument at index 0, which they can use to access public, private and protected instance members of the enclosing type. When an instance method is called, the stack must contain the arguments preceded by the instance pointer.

A method is identified by its name, class type, and signature. When calling a static method, the type of the class is needed. And when calling an instance method, an instance of a class type needs to be provided. The signature exists of the return-type of the method, the number of arguments and the argument types. When a method is called, the CLR searches for a method containing the same name, type and signature as provided in the call. As soon as a matching method is found, the arguments (which should be placed on the stack prior to calling) are copied from the stack to an array that holds the passed arguments.If the `init` directive is present, the local variables are initialized to the type's default value. For example a variable of value type `int32` is initialized to the value 0. If the `init` directive is not present, is deemed unverifiable in a security check performed by the CLR [15]. After initialization of the local variables, the method's evaluation stack is empty and the execution of the first instruction can start.

When the method reaches the last instruction, which is the `ret` statement, the return value (if available) needs to be on the evaluation stack, and the method state transfers control to its caller.

2.2.5 The IL Instruction Set

The IL instruction set provided in Partition III of the CLI Specification [6] is partitioned into two sections, called base instructions (e. g. addition and subtraction) and object model instructions. There are over 220 instructions. A full documentation of the IL instruction set can be found in the CLI specification [6].

Most IL instructions perform their actions by using the evaluation stack that is associated to each method state (see Figure 2.6). For example an add expression that adds two values *value1* and *value2* yields a *result*. What happens in IL is represented in Figure 2.9. The two values are pushed on the stack by using the `ldc` instruction. Subsequently an arithmetic operation (`add`) is executed, which pops the two values from the stack and replaces them with the resulting value. Note that *value1*, *value2*, and *result* represent actual values.

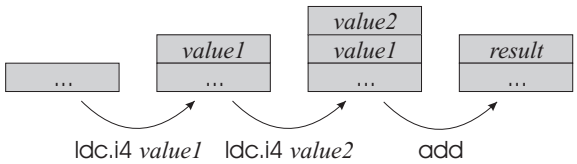


Figure 2.9: Execution of instructions on the stack

The .NET Intermediate Language contains instructions that are completely independent of the type of their arguments [19]. For example, it is possible to use the same instruction to load a value of a local variable on the stack for both an integer and a floating-point number. The reason for this design decision is that Microsoft wanted the creation of source-to-IL compilers to be as easy as possible, in order to extend multi-language support.

Sometimes IL instructions are used with an efficient encoding. For example, for loading an argument onto the stack, it is possible to either use the instruction `ldarg <num>` (for which an *int16* number represents the index), or the instructions `ldarg.0`, `ldarg.1`, `ldarg.2` and `ldarg.3` which are encodings for the most often used arguments of a method. The first instruction needs an extra two bytes of memory for the index, while the other instructions have the index encoded in

the instruction. Furthermore, the CLR does not need to read the instruction argument, resulting in some performance gain. For indices 4 to 255 it is also possible to use `ldarg.s` followed by an *int8* number representing the index, which is called a short form.

The IL instruction set can be categorized further than the previously mentioned categories base instructions and object model instructions, as presented in the following sections. For a full and detailed list of instructions, we refer to [6].

Load and store instructions

These are instructions used to load values or references onto the stack and retrieve them from the stack to store them at their home locations. Typical examples of such instructions are `ldarg`, `ldloc`, `ldobj` and their counterpart instructions, being `starg`, `stloc` and `stobj`.

Arithmetical, logical and type conversion instructions

To be able to support arithmetical operations, IL contains typical arithmetic instructions such as `add`, `sub`, `mul` and `div`. IL also supports logical instructions (called bitwise instructions in the IL specification[6]) like `not`, `and` and `or`. An example of a type conversion instruction is `conv`, which is available to convert the value on top of the stack to the specified type.

Branching instructions

In IL there are a number of instructions that are used to control the flow of execution. We distinguish conditional and unconditional branch instructions. Conditional branch instructions either take one value from the stack (and check whether a condition specified by the used instruction is true) or they compare two values on the stack. Depending on the outcome of the condition a branch follows. For example, the instruction `brfalse` adjusts the control flow if the value on the stack is `false`. Another example is the instruction `beq` which stands for 'branch on equal'. In this case the control flow is adjusted to a target if the top two values on the stack are equal. In both cases, the program does not branch and continues executing the next instruction if the condition is not satisfied.

Unconditional instructions are instructions that do not depend on a condition. An example of such an instruction is `br`, that unconditionally branches to a specified target.

Miscellaneous instructions

Beside previously mentioned instruction types, IL also contains instructions like calls and a return instruction.

There are different types of call instructions in the IL. `call` is used for calls to static methods of which the destination is fixed at compile-time, while `callvirt` uses the class of an object (known at runtime) to determine the method to be called. The `callvirt` instruction is used for both instance methods.

The return instruction `ret`, which is used to return from a method, is performed without any condition. The value that is on the evaluation stack, if there is any, is copied to the evaluation stack of the caller and control is transferred to the caller.

2.2.6 Generics

Generics allows defining a class or method without a specific type. The defined item can then be reused with several types. Generics provides type safety at compile-time.

The generics concept is introduced in version 2.0 of the .NET Framework, but is not implemented in our translator.

2.2.7 Name Resolution

Names in the IL exist of a *simple name* or of a composition of simple names with connection symbols such as a dot. For example, `System` and `Object` are simple names, while `System.Object` is a composite name. A composite name is also called a *dotted name*.

The common prefixes of *full class names* are called *namespaces*. The full name of a class is a dotted name. The last simple name of the dotted name is the class name. For example the dotted name `System.Object`. Here `Object` is the class name and `System` is the namespace.

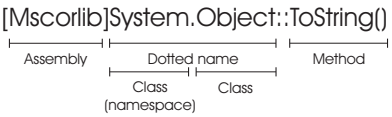


Figure 2.10: A method call, referenced by its assembly and dotted name.

A class is scoped to a particular namespace, and a namespace is scoped to the provided assembly. If no assembly is provided, the namespace is scoped to the current assembly.

2.3 Our Work

In the rest of this report – especially in Chapter 4 and Chapter 5 – we treat the .NET concepts that we have implemented. These cover the implementation of a stack representation, support for integer types and object types, and the ability to instantiate objects and call both static and instance methods. The instructions we treat are involved with arithmetical operations as well as instructions that do comparing and branching. We also implemented support for instructions that are used to explicitly load and store values from and to arguments, fields, locals, and the stack.

2.4 Summary

In this chapter we have introduced the Microsoft .NET Framework. We have recalled what Microsoft's objectives are for developing the .NET Framework, and which components the framework consists of. We have explained that the Common Language Runtime is the environment in which .NET applications are executed. It uses IL code as input and compiles this to native code, prior to executing it. This process is called JIT compiling. Besides the execution of IL by the CLR, we have introduced the Common Type System and Common Language Specification.

Furthermore, we told something about how IL is stored in a binary format in a so-called Portable Executable file. This file is loaded by the Virtual Execution System, which is part of the CLR. We also have introduced some aspects of code management, which covers the usage of a stack and heap. The stack is used by the CLR to store intermediate values and references on. The heap is a dynamic storage area in which objects of classes and arrays can be stored. When executing a program, each method gets a method state assigned. A message state contains information about the environment within which the method executes, like an instruction pointer and lists of input arguments and local variables.

The Intermediate Language was also introduced in this chapter. We have told something about directives, which are commands that ask the runtime-environment to perform a task, as well as about modules and assemblies. Modules are physical files that can be embedded in an assembly, which is a logical unit of functionality. The IL also uses namespaces, which can be used to group functionality within unique names. We also mentioned methods, which are operations associated with a type, and what happens when they are called. There are two type of methods, namely instance methods and static methods.

The IL instruction set contains over 220 instructions that can, among others, be used to load and store values or references on the stack, perform arithmetical or logical operations, do type conversions, and adjust control flow. Furthermore we have also mentioned how names and name resolution look like in IL.

Chapter 3

Graphs and Graph Transformations

Graphical structures like charts and diagrams, are often used to represent complex data and structures in an intuitive way. A graph is such a graphical structure, and is applied in different areas like route planners, electric circuits, job scheduling and train-networks.

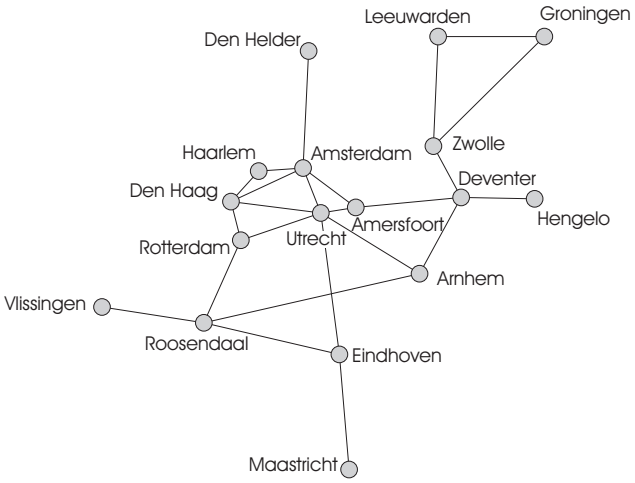


Figure 3.1: A graph representation example

Figure 3.1 is an example of a graph. It represents a number of road connections between cities in The Netherlands, the nodes representing cities and the edges being the roads. The cities contain labels with the name of the city. In this example we omitted the labels on the edges. However, edges could be labelled with names (of the roads) or values (representing distance, fuel usage, travel time, travel expenses, etcetera).

In this research project, we use graphs to model the compile-time and run-time structures of a program. A compile-time structure can be a concrete or abstract syntax representation of an arbitrary program. The run-time structures of a program are state snapshots of the program while being executed. Graphs are useful for this because they have a formal background, are intuitive and can be used for modelling many different application areas. We think that graphs are useful for representing the compile-time structure of a program, as well as the run-time behaviour of a program involving dynamic (de)allocation of storage space and dynamic method invocation. The behaviour of software programs is simulated by (repeatedly) transforming one graph into another. To transform one graph into another graph we use *graph production rules*. Production rules are described in Section 3.2.

Throughout this chapter we use Pacman examples to explain different concepts. These examples are based on the examples presented in [8].

3.1 Graphs

A graph is a mathematical structure. We use edge-labelled graphs defined over a set *Lab* of labels, as follows [12]:

Definition 3.1 (Graph)**.** A graph G is a tuple $\langle Nod, Edg \rangle$ where

- Nod is a finite set of nodes;
- Edg $\subseteq$ Nod $\times$ Lab $\times$ Nod is a (finite) set of edges.

The graphs we use are directed graphs, i. e. for each edge we distinguish between its *source* and *target* node. Furthermore, as follows from the definition, edges have a label and nodes can not. It is possible to create an edge with the same source and target node, i. e. self-edges of a node. Self-edges can be considered as a way of labelling nodes. A node can have multiple self-edges and thus multiple labels. In this setting, it is not possible to have more than one edge with the same source, target and label, i. e. parallel edges.

3.1.1 The Pacman Example

An example graph, bases on the Pacman game, is given in Figure 3.2. In the graph a number of nodes are shown, namely the dots, and the figures of Pacman, the ghost and the apple. The grid of dots and the edges between them represent the fields to which Pacman, the ghost and the apple are bound. The normal behaviour would permit both Pacman and the ghost to be able to move over the grid. To keep this example simple, we assume the ghosts to be fixed to a specific node in the grid. For simplicity, we also omitted the labels on the edges.

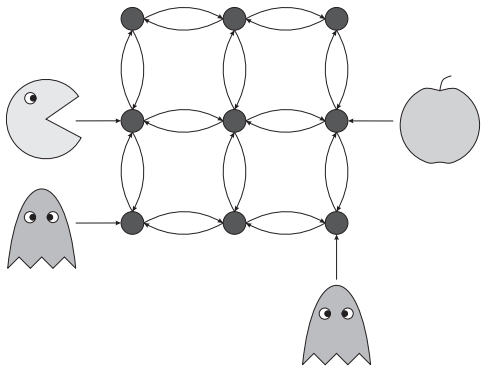


Figure 3.2: Graph representation of Pacman

3.2 Graph Production Rules

To transform a graph (source graph) into another graph (target graph), we use graph transformation rules. Graph transformation rules are also called *graph production rules.*

A graph production rule p has the form $p : L \to R$, in which L represents the left hand side (LHS) graph and R the right hand side (RHS) graph.

A match for $p : L \to R$ in some graph G is a total morphism $m : L \to G$, i. e. the occurrence of p's LHS in G. Applying rule p means finding a match of L in the source graph G and replacing

L by R, leading to the target graph of the graph transformation[7, 8]. This replacement is not complete, because the structure is preserved wherever the L and R overlap[13].

Application of a production rule can be written as $G \overset{p,m}{\Rightarrow} H$, meaning that graph G is transformed into graph H by using production rule p at matching m. It is possible that p has multiple matchings of its LHS in graph G, but also that multiple rules are applicable to the same graph.

Production rules can be extended with *negative application conditions* (NACs, see [7, 10]). A negative application condition limits the applicability of a rule by extending the rule's LHS. A rule p will only be applied to a source graph G when the LHS matches G and if that matching cannot be extended to a matching of any NAC of that rule.

3.2.1 The Pacman Example - Production rules

An example of a production rule in the context of the Pacman game is presented in Figure 3.3. According to this rule, called *move*, Pacman moves to a new position by removing the edge between Pacman and a node n (e. g. the left node in the rule below), and placing an arrow between Pacman and the neighbour node of n.

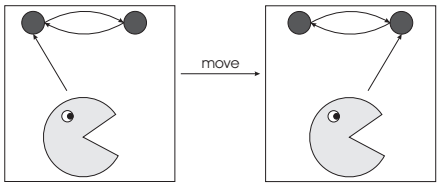
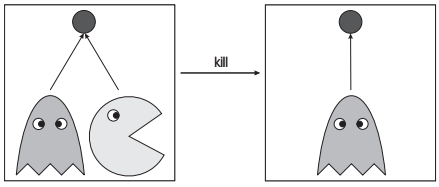
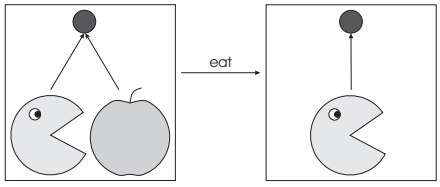


Figure 3.3: Production rule to move Pacman

Of course it is also possible to simulate the behaviour of a ghost eating Pacman, or Pacman eating an apple. See for example, Figure 3.4.



(a) Rule of Ghost killing Pacman



(b) Rule of Pacman eating an apple

Figure 3.4: Two additional production rules

In Figure 3.5 the application of a production rule is displayed. For this example, we use the already introduced production rule for moving Pacman and the graph representing Pacman and the grid of nodes. (For space reduction, we have presented only a part of this graph.)

In this example, we have a production rule p consisting of left hand side L and a right hand side R and a transition between L and R. There is a matching m between L and the source graph G. This is indicated with the dashed and dotted arrows. Now that there is a matching m of rule p, it is allowed to transform graph G by replacing the matched nodes of L by the nodes of R, resulting in a graph H. Note that the arrow in graph G, denoting the position of Pacman, is deleted and that in graph H a new arrow is constructed. Thus, Pacman has been moved from one position to another.

As mentioned before, production rules can be extended with NACs. In Figure 3.6 we extended the previously introduced rule to move Pacman from one node to another (see Figure 3.3) with a NAC. In this example, Pacman may only move to a node when there is not a ghost positioned at that same node.

To execute this rule, a matching m between the LHS of the rule and the source graph must exist. Furthermore, the elements of the NAC must be excluded from the source graph. If this is

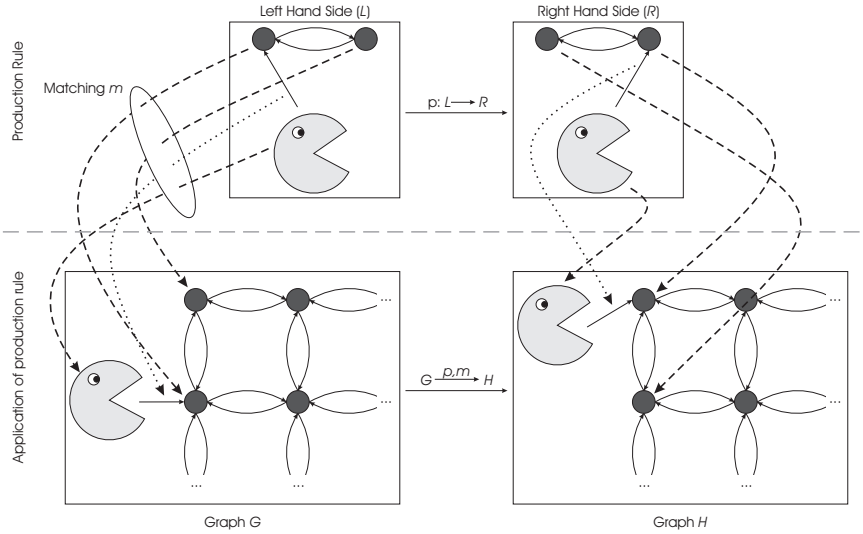


Figure 3.5: Example application of production rule

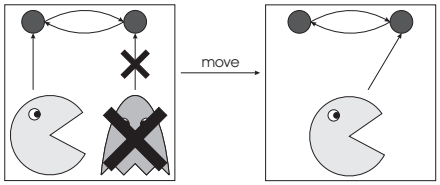


Figure 3.6: Production rule to move Pacman, containing Negative Application Condition

the case, the NACs are satisfied and the rule can be applied by replacing the matched elements of the LHS by the elements of the RHS of the rule.

3.3 Graph Production System

A *graph production system* (GPS) consists of a set of graph production rules $\mathcal{R}$ and a start graph I. The GPS can be used to generate a (possibly infinite) state space by applying production rules $p \in \mathcal{R}$ to the graphs, starting with graph I. All the resulting graphs can be seen as state snapshots (states), and the application of the rules as transitions between the states. The set of graphs and transitions between these graphs, we call a *graph transition system* (see also [14]). A graph transition system always contains an initial state. Furthermore, it can have intermediate states and a final state if the state space is finite.

3.3.1 The Pacman Example - Graph Transition System

In the context of the previously introduced Pacman example, we present an explanation of a graph transition system. For this example, we have taken the graph presented in Figure 3.2 as start graph. When applying the rules from Figure 3.3 (*move*) and Figure 3.4 (*kill* and *eat*) whenever possible, we get the graph transition system displayed in Figure 3.7.

Note that a state can have transitions from one state to another (and possibly back). Each transition represents the application of a production rule (*move, eat or kill*). The two highlighted states match with the source and target graph the *move* production rule as explained in Figure 3.5. Furthermore, we can see that once the apple has been eaten, there is no way back to a state having an apple positioned at the grid. Also, we can see that once Pacman has been killed by a ghost, there are no applicable rules left.

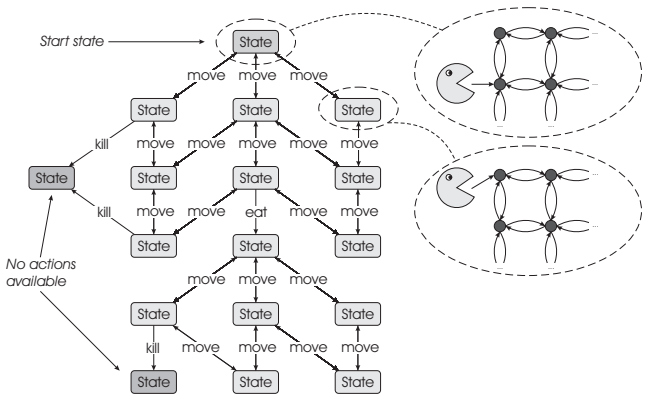


Figure 3.7: Transition system of Pacman example

3.4 Graph Transformation Tool

The tool we use for performing graph transformation is GROOVE [22]. GROOVE stands for GRaphs for Object-Oriented VErification. With this tool it is possible to specify graphs and production rules. It can also execute these production rules, which results in a graph transition system.

In GROOVE, graphs are represented by blocks for nodes and arrows for edges. Self-edges are represented as an arrow with the same start and end node, or as a label on a node (inside the rectangle).

Furthermore, GROOVE offers the LHS, RHS, and the NACs of a production rule to be represented in a single graph. To accomplish this, the production rules used by GROOVE consist of four different types of nodes and edges, each having different shapes and colours [21, 13]:

- *reader*-elements are elements that occur in both LHS and RHS. They have to be present in the source graph to match the LHS and are preserved in the target graph. Reader-elements are represented by thin solid black arrows and rectangles.

- *eraser*-elements are elements that occur in the LHS but not in the RHS. They have to be present in the source graph to match the LHS, but are deleted in the target graph. Eraser-elements are represented by thin dashed blue arrows and rectangles.

- *creator*-elements are elements that do not occur in the LHS but do occur in the RHS. They have to be absent in the source graph in order to be introduced in the target graph. Creator-elements are represented by thick solid green arrows and rectangles.

- *embargo*-elements are elements that prohibit the application of the rule when they exist in the relating matching in G. Embargo-elements are making up the NACs and are represented by thick red dashed arrows and rectangles.

3.4.1 The Pacman Example - GROOVE

An example production rule used in GROOVE is displayed in Figure 3.8. In this figure, which is based on the rule presented in Figure 3.6, it can be seen that this rule only is applicable if Pacman and two adjacent nodes are available in the source graph. Pacman also must be at the node the eraser-element points to. If this is the case, the embargo-elements are checked. The embargo element for this rule states that no Ghost may be positioned at the node adjacent to the node Pacman is positioned at. If so, the rule can be executed; the eraser-element is deleted and the creator-element is created.

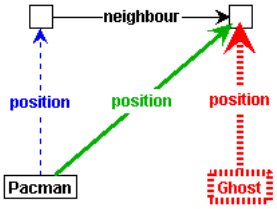


Figure 3.8: Example production rule in Groove

3.5 Summary

This chapter introduced the concepts of graphs and graph transformations. We have told that our graphs contains nodes and labelled edges. Furthermore, we have mentioned that it is not possible to have parallel edges. A (source) graph can be transformed into another graph (the target graph) by using graph production rules. A graph production rule can be applied to a source graph when the left hand side of the production rule has a matching in the source graph, but only when negative application condition are satisfied.

Furthermore, we have provided a brief description of the graphical notation of graph transformations in the GROOVE tool set. There, we distinguish four types of nodes and edges, namely: reader, eraser, creator, and embargo elements.

In this research project graphs are used to model compile-time and run-time structures of a program, while graph transformations are used to represent the behaviour of the program.

Chapter 4

Translating IL Programs to Graphs

Before introducing the developed graph production rules in the next chapter, we focus on the start graph to which the graph production rules are applied. This start graph is an abstract model of the IL program to be simulated. Because the .NET language compilers generate IL bytecode, and the GROOVE tool set needs a graph as input, a translator is developed that translates arbitrary IL programs into a graph representation of that program. Such a graph is called an Abstract Syntax Graph (ASG).

This chapter starts with a description of the translator. We will introduce the structure of the translator, what its input is and which operations are performed on this input. In Section 4.2 a meta-model of the ASG is shown and discussed.

In Section 4.3, we propose a representation for namespaces. We also discuss how method signatures are calculated, how they are represented in the graph, and how existing method signatures are resolved. Additionally, Section 4.3 contains a description of the static analysis process performed by the translator.

After discussing static analysis, we present an example in which we translate two equivalent programs written in different .NET languages (i. e. C# and VB.NET) to the .NET IL. In this example, we will show that both programs will result in two IL programs having comparable semantics. Furthermore, we will provide an example of the translation of an IL program to an ASG.

4.1 Translator

The translator uses a textual IL program as input. This textual IL program is obtained by disassembling a Portable Executable file (see Section 2.1.5), using the disassembler called `ildasm`. The tool `ildasm` is incorporated in the .NET Framework SDK 2.0, which is freely available from the Microsoft website[1]. Because we are disassembling a program that has already been type checked by the compiler that created the PE file, we can assume that the program is type correct. Note that this only applies to type errors detectable at compile-time and not at run-time (such as explicit type casting of objects). Because we assume that programs are correctly typed, we do not perform any type checking in neither the translator nor the production rules.

During the translation phase, a textual IL program is read and transformed into a graph representation of this program. To do this, we could implement our own translator from scratch or use a compiler generation tool that creates one for us. Implementing a translator from scratch would involve a lot of work. Therefore we have chosen to use a compiler generator tool called ANTLR[2]. ANTLR uses grammar specifications as input and automatically generates a translator

[1] Download .NET Framework SDK 2.0 from: `http://msdn2.microsoft.com/en-us/netframework/aa731542.aspx`

according to this specification. We obtained a grammar for a .NET Intermediate Language *parser* (written by Pascal Lacroix) from the ANTLR website, which had to be modified due to non-determinism. The grammar file for the *tree walker* is our own implementation. The generated translator consists of a *lexer*, a *parser*, and a *tree walker*.

Lexer The lexer scans the input file (i. e. an IL program) and chops it into pieces called tokens. These tokens are sequenced into a stream – called a tokenstream – and sent to the parser.

Parser The tokenstream is used as input for the parser which creates an Abstract Syntax Tree (AST). An Abstract Syntax Tree is a tree-shaped abstract representation of the program. The AST can be used to transform and order program information. Furthermore, it is used to omit syntactic information from the original program without losing its semantics. This makes it easier to process it further by the tree walker.

Tree walker A tree walker is used to visit all nodes in the AST and to create a new structure in the form of an Abstract Syntax Graph.

An overview of the translation process is presented in Figure 4.1. The rectangles represent the input and output of the different parts of the translator. The labels on the edges denote the parts of the translator that are responsible for translating an input into an output.

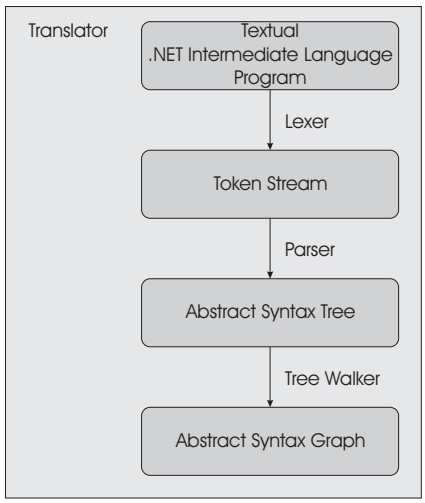


Figure 4.1: Overview of the translator

The result of the translator, the Abstract Syntax Graph, is used as GROOVE's start graph for simulation using graph transformations. This is explained in Chapter 5.

4.2 Meta-Model Abstract Syntax Graph

In order to formalize and give an overview of the structure of the ASG, a meta-model has been designed. Because this meta-model is too large to fit in one figure, it is spread over multiple figures, which are Figure 4.2 to Figure 4.5. Together, these figures describe the concepts that are used in the ASG, and the relations between these concepts.

4.2.1 High-level structure

We start with a meta-model containing the overall structure of the ASG. This model is shown in Figure 4.2.

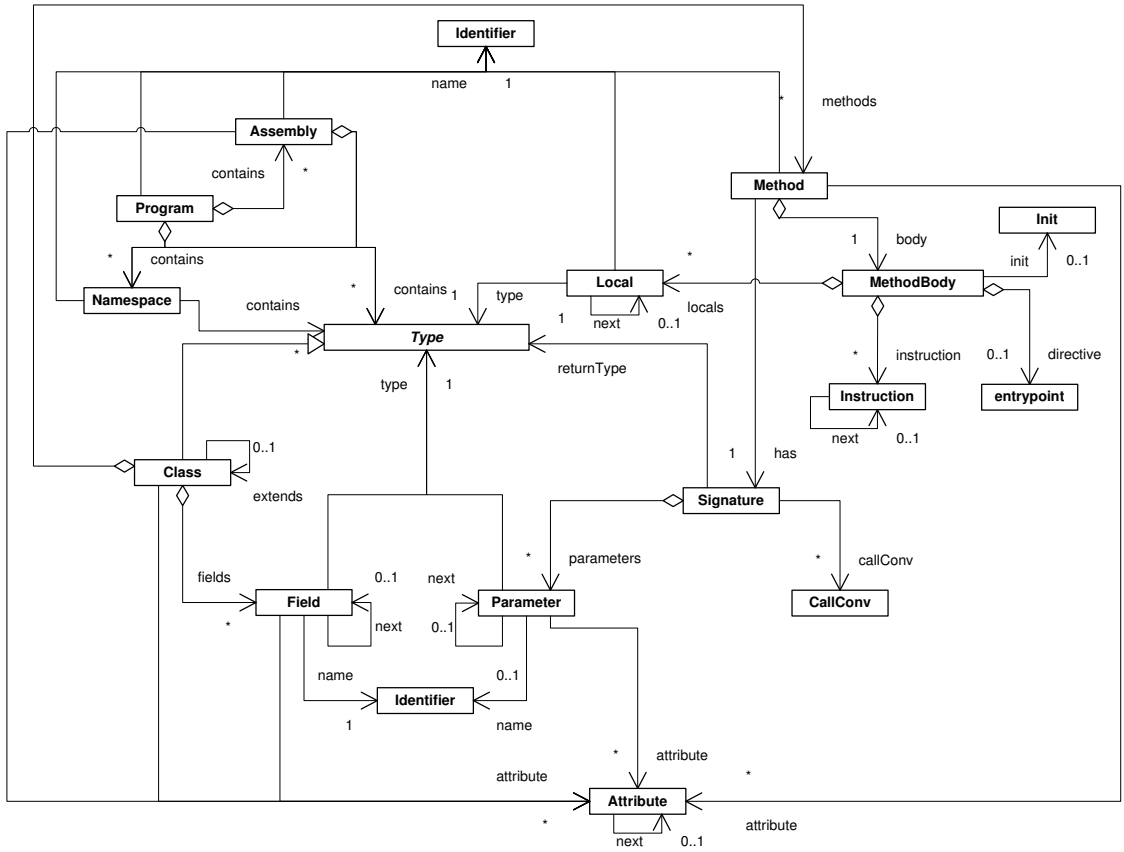


Figure 4.2: Meta-model of the Abstract Syntax Graph

The rest of this section contains a description of the concepts presented in the meta-model presented in Figure 4.2. Note that this contains two Identifier nodes. Both Identifier nodes represent the same concept of an identifier and were put in the model for preventing too many crossing lines.

Program The Program concept is the root of the graph and represents a parsed IL program. A Program can contain Namespaces, Types and Assemblys.

Namespace A Namespace represents the logical grouping of the names used within a program. They can be nested in an IL program, but we choose to represent this in a non-nested manner, because classes are always referenced by its fully qualified name. For more information see Section 4.3.1. A Namespace can contain Types, in the form of Classes.

Assembly The Assembly concept represents a logical unit that can hold Classes and Namespaces. Furthermore, an Assembly is identified by a name and can have Attributes. We use the Assembly concept (at this moment) only to determine whether or not classes from the .NET Class Library are called.

Type The Type concept represents types like classes and value types. See Section 4.2.2 for more information.

Class The Class concept is a subtype of the Type concept and represents a class declaration. A Class can extend another Class, has a name (which is identified by its Identifier) and can contain Fields and Methods.

Signature The Signature node is used to represent method signatures. A method signature contains the method's return type, the calling convention, and the method's parameter types in an ordered way. The signature is determined by the parser; how this is done will be explained in Section 4.3.2.

Method The Method node represents a method declaration. A method declaration always has a Signature and must have a MethodBody.

MethodBody MethodBody represents the method's implementation.

Instruction Instructions are denoted by the Instruction concept. Section 4.2.4 discusses the individual instructions.

Identifier The Identifier node represents a name that is used to name an element.

CallConv CallConv represents the calling convention of a method. The calling convention denotes whether or not the call is to a static method.

Parameter The Parameter concept represents a formal parameter in a method signature and describes variables which are accepted by a method. Parameters are of a certain Type, may have a name (identified by the Identifier concept), and are contained in the method Signature.

Field The Field node represents the declaration of a field. A Field has a name (identified by Identifier), attributes (denoted by Attribute), and has a type relationship to the Type to denote its type.

Local The Local concept represents the declaration of a local variable. A Local can have a name (identified by Identifier), and has a relationship to the Type to denote its type.

Init The Init concept is used to denote that the Local variables of a MethodBody must be initialized to their default values.

Attribute The Attribute concept represents available attributes. It is further explained in Section 4.2.3.

Entrypoint The entrypoint concept denotes the start point of executing a Program.

4.2.2 Types

This meta-model (see Figure 4.3) describes the types that are available in the ASG. The following types are supported in our ASG and production rules.

bool The bool concept represents the type bool (i. e. boolean) of which the values can be either a true (non-zero) or false (zero).

char A value of type char can hold a single Unicode character.

string The string concept represents the type string. An instance of type string is a sequence of characters.

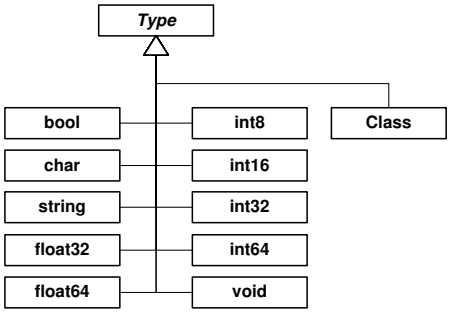


Figure 4.3: The types in the ASG meta-model.

float32, float64 The concepts float32 and float64 represent floating-point types (which describe floating-point values). Note that this concept is not used (yet), because GROOVE does not support floating point values.

int8, int16, int32, int64 These nodes represent integer types of different sizes (respectively 1, 2, 4, and 8 bytes). We do not distinguish between these types during run-time simulation using production rules, because we do not have a notion of memory-sizes.

void The concept void is only used as a return type, indicating that a method does not return a value.

Class The Class node represents a class type which can contain Fields and Methods. See Figure 4.2.

4.2.3 Attributes

The meta-model presented in Figure 4.4 contains a description of the different kinds of attributes available. These attributes are at this moment not used in the production rules, but we have put them in the graph for possible future extension.

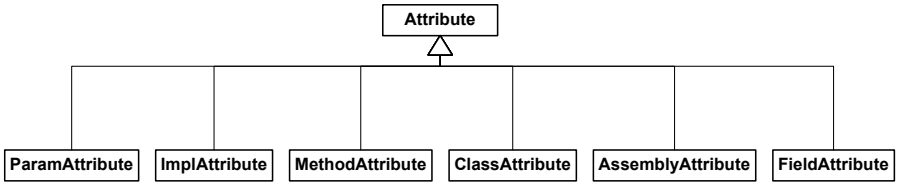


Figure 4.4: The attributes in the ASG meta-model.

Examples of well known attributes are `private`, `public`, `static`, and `abstract`. For all the different types of attributes, we refer to the ECMA specification [6].

4.2.4 Instructions

The instructions denoted by this meta-model are the .NET IL instructions that a method body can hold. The instructions we support are given in Figure 4.5.

Although we have presented all supported instructions in the meta-model, we do not discuss all instructions individually. However, we would like to highlight a few important and interesting instructions. Note that IL instructions have a strong resemblance to instructions in assembly languages.

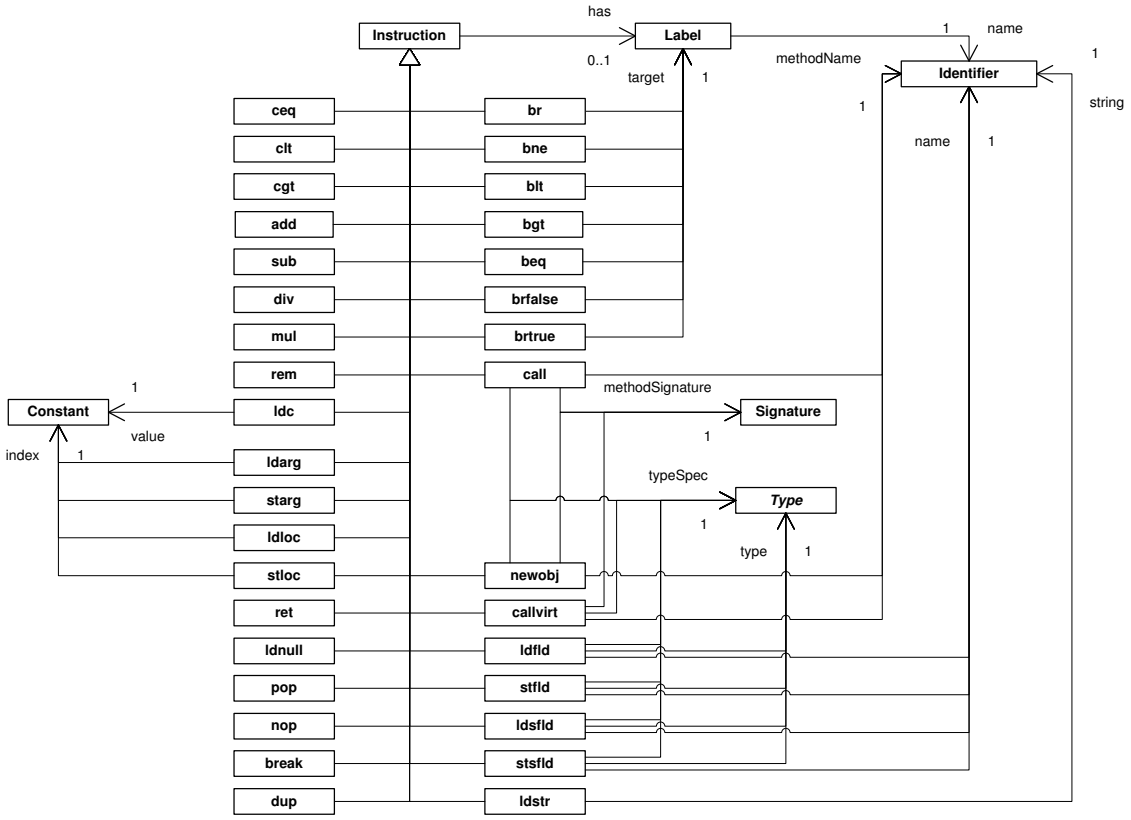


Figure 4.5: The instructions in the ASG meta-model.

Instruction See Section 4.2.1.

Label The Label concept denotes a label that is used to tag an instruction, which can be used as a target to branch to. In this project a label always is represented by an Identifier.

ceq, clt, cgt The ceq, clt, and cgt instructions stand for compare if equal, less than, and greater than. When executing these instructions, two values on the stack are popped, compared, and the result of this evaluation is pushed back on the stack. The result is 1 (of type integer) if the evaluation yields true, otherwise 0.

add, sub, div, mul, rem These arithmetical nodes represent arithmetic instructions, namely addition, subtraction, division, multiplication and remainder (modulo).

ldc The instruction ldc loads a Constant value on the stack.

call, callvirt, newobj The concepts call and callvirt represent static and dynamic calls, respectively. The instruction newobj represents the creation of a new object. All three instructions are related to the Signature node, representing the signature of the method to call. In case of the newobj instruction this is the constructor's Signature. Furthermore, the instructions use an Identifier representing the name of the method to call. Also, they are related to a type which represents the class of which the method must be called, or – in case of the newobj instruction – of which an object must be instantiated.

ret The ret instruction is used to return from a method to its caller. In this project we assume that a method is always ended with a retinstruction. In IL there are other possibilities to terminate a method, i. e. with `throw` or `jmp`, but these are not supported yet.

ldfld, stfld The ldfld instruction is used to load a value from an instance field to the stack, and the stfld instruction is used to store a value from the stack to an instance field. Both the ldfld and stfld instructions use an edge with label typeSpec to refer to the type of the instance of which the field that the value that is loaded or stored is an instance. The name edge points to an identifier representing the name of the field. Furthermore, the instructions have an edge called type to a specific Type to indicate the type of the field.

ldarg, starg The instructions ldarg and starg are used to load the value of an argument on the stack and store a value from the stack to an argument, respectively. The argument is indicated by a Constant value representing the index of the argument.

ldloc, stloc The ldloc and stlog instructions are used to load a value from the stack to a local variable, and vice versa. The used local variable is indicated by its Constant value representing the index. The typeSpec edge indicates the type of which the object is an instance of. The name of the field is referred to by an Identifier node.

br, bne, blt, bgt, beq, brfalse, brtrue These instructions represent branch operations. The instruction br represents an unconditional branch to an instruction bound to a specific Labelrepresenting its target. The other instructions depend on the evaluation of a condition. If the evaluation yields `true`, then the control should branch to the target-instruction represented by the Label. Otherwise, the control is transferred to the next instruction.

Type See Section 4.2.2.

Identifier See Section 4.2.1.

Signature The Signature concept represents a method signature and is used to locate a method implementation.

Constant The Constant concept stands for GROOVE's way of representing an actual value of a specific type, which can only be integer, boolean or string.

4.3 Design Decisions

Implementing the translator came along with a number of problems. Decisions about the representation of namespaces, classnames, signatures and identifiers had to be made. These design decisions are discussed in this section.

Furthermore, we do not only discuss the representation of these concepts, but also describe the process of static analysis in this chapter. During this static analysis phase, namespaces and classnames are transformed in order to meet their specified representation to be able to use them in graph production rules. We also describe how method signatures that are not explicitly available in the program are computed and stored in the final graph. Furthermore, the process of resolving identical string values of identifiers is part of static analysis.

4.3.1 Classnames and namespaces

In the IL it is possible to have nested namespaces, which we will explain on the basis of the example code of Listing 4.1. We have left out the details in this pseudo code in order to emphasise the namespace structure.

The example shows that it is possible to have a namespace `A.B`, which stands for a nested namespace `B` in namespace A. Furthermore, a namespace `E` is nested in a namespace `D` to create the nested namespace `D.E`. Also, classes can be declared in a namespace. Classes `X`, `Y`, and `Z` in the example are declared in the namespaces `D` and `D.F`, respectively.

```
1  .namespace A {
2    // This is namespace A
3  }
4
5  .namespace A.B {
6    // This is namespace A.B
7  }
8
9  .namespace A.C {
10   // This is namespace A.C
11 }
12
13 .namespace D {
14   // This is namespace D
15
16   .class X {
17     // This is a class with the full name D.X
18   }
19
20   .namespace E {
21     // This is namespace D.E
22   }
23
24   .namespace F {
25     // This is namespace D.F
26
27     .class Y {
28       // This is a class with the full name D.F.Y
29     }
30
31     .class Z {
32       // This is a class with the full name D.F.Z
33     }
34   }
35 }
36
37 .namespace G.H.I {
38   // This is namespace G.H.I
39 }
```

Listing 4.1: Namespace IL example

A useful namespace representation in the ASG is needed in order to use namespaces in production rules. Therefore, we propose three alternatives.

Alternative I. Nested namespaces (Figure 4.6(a))

For the first alternative we use a real nested structure as one can encounter in a program. Each namespace node represents one single name of the full namespace. Thus, as present in our example, the namespace `A.B` would consist of a namespace node `A` as child from `Program`, and a namespace node `B` as child from namespace node `A`. The advantage of this representation is that we maintain the nesting structure. On the other hand, using this representation, it is quite hard to directly

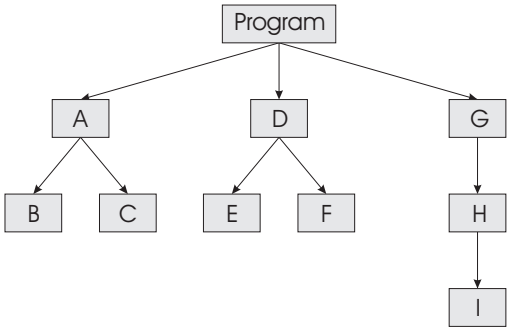
resolve a nested namespace by its full name. For example, in this representation resolving the namespace `A.B` by using production rules would involve multiple production rules to resolve each (sub)namespace.

Alternative II. Flat nested namespaces (Figure 4.6(b))

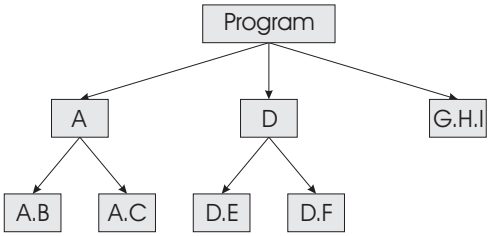
This representation also uses a nested structure of namespace nodes, but this time with dotted name. This has as advantage that it preserves the nested structure, and that it is possible to resolve namespaces directly by their full name. For example, in Listing 4.1 we declare a namespace `G.H.I`, without using the namespaces `G` and `G.H`.

Alternative III. Flat namespaces (Figure 4.6(c))

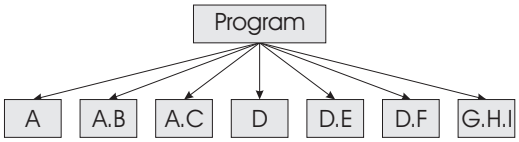
This representation uses dotted names for the namespaces, and relates them directly to the Program node. Thus, nesting information is not explicitly available. This is not a problem because in IL no relative names are used without using a fully qualified name. This representation is easier to implement, because every namespace node can simply be attached to the Program node.



(a) Single names, nested



(b) Dotted namespaces, nested



(c) Dotted namespaces, attached to Program

Figure 4.6: Namespace representation proposals.

We have chosen for the third alternative and will represent namespaces in the ASG by using dotted names, attached to the Program node. To get this result in the ASG for an arbitrary program, the translator must transform the namespaces and classes to this representation into full names (in case these names are not already in this representation).

The translation of the namespaces and classes is performed in the parsing phase (Section 4.1). When the translator encounters a namespace declaration, it looks if a higher-level namespace was already declared (i. e. if the namespace is nested within another namespace) and stores a combination of the higher-level namespace (if available) and the new one both in memory and in the Abstract Syntax Tree. The same holds for classnames. When the parser encounters a classname, it looks if a namespace was already declared for this scope. This means that the classname is nested within a namespace and that the classname has to be combined with the namespace.

Transforming the code of Listing 4.1, results in the graph represented in Figure 4.7. Note that this still is a stripped down graph, merely to explain what the namespace structure looks like.

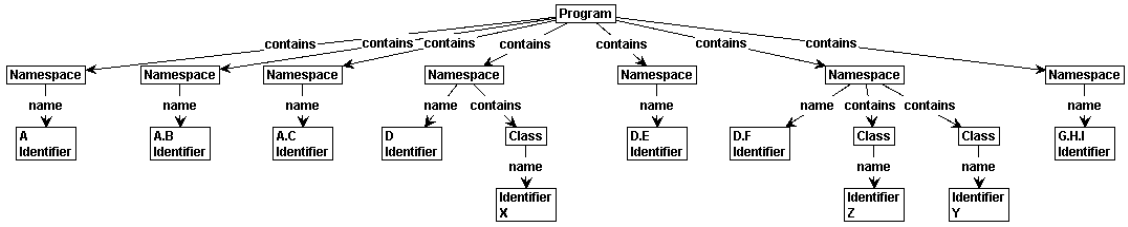


Figure 4.7: ASG of Listing 4.1.

What we did not mention up to now is that classnames and namespaces are stored in the Abstract Syntax Tree in order to be able to create method signatures (as described in the next section). Method signatures are used for resolving method calls during the simulation phase.

4.3.2 Method signatures

A signature of a method is defined by its calling convention, its return type, and the number, order, and types of the parameters. Note that a method signature in IL does not contain the name of the method. This is for example in contrast to Java, where the signature contains the name of the method along with the number and types of the parameters (and their order). Methods are compatible when they share the same signature. But when a method is called, the method lookup is performed by using the method name combined with the corresponding signature as specified in the call.

Our solution is to create unique method signature nodes, and references to these nodes, by the translator. After determining the signature, a lookup is performed on a set containing references to existing signatures. If the signature already exists, an edge is created to the node representing this signature. If the signature does not exist, we create the nodes representing the signature, store it in the set of signatures, and create an edge to the node representing the signature.

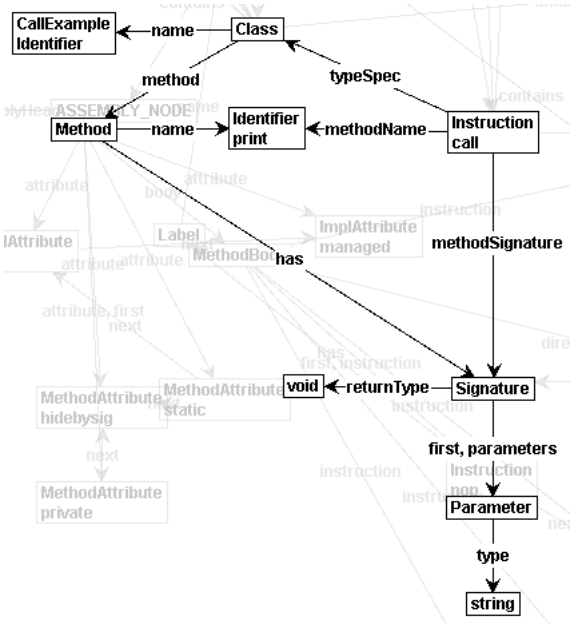


Figure 4.8: Graph representation of a referenced signature.

Figure 4.8 contains an example of how signatures look like and how they are referred by method calls. The class, named `CallExample`, contains a method with the name `print`. For this method, a signature is created with `void` as return type. Furthermore, the signature shows that the method `print` accepts a string as parameter. The figure also contains a call instruction that calls the method `print` of class `CallExample` with a specific signature, namely the method with return type `void` and a parameter of type `string`. Now that the signatures are resolved, it is possible to find the correct method implementation during simulation.

4.3.3 Identifiers

Identifiers values, such as names, are represented by one node for each unique denotation. Thus when two labels have the same name, this is represented by two label nodes both pointing to one single node with the identifier name as its label. This is accomplished by generating a key from the identifier name and determining if, according to this key, the identifier is already known. If it is unknown, the node representing the identifier is created. Otherwise the existing node is used.

For example, instructions can contain labels (which are equivalent to identifiers) and can target to labels. When these labels have the same name, they must point to the same node (see Figure 4.9).

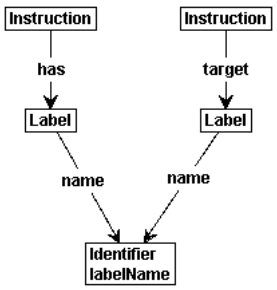


Figure 4.9: Relation of labels to identifiers

The idea of this representation is that in the code of the program physically two labels are present, but they both contain the same name. Thus, in our representation we use two label nodes, but only one single name node. During static analysis with production rules, these labels are resolved. Off course, it is possible to identify the labels (instead of their identifiers) in the translator, but we have chosen to use graph production rules in order to provide more intuition in what happens during this process.

4.4 Translating C# and VB.NET to IL

In the introduction we claimed that when we cover semantics of the IL instructions, it should be possible to simulate programs written in every .NET language. To support this idea, we have written two simple programs. The first is written in C#, the second in VB.NET. Both programs were compiled to an executable and disassembled. For the C# program, we used the following command line options:

```
csc /t:exe /optimize+ program_name.cs
ildasm /text /out=program_name_cs.il program_name.exe
```

And for the VB.NET program we used:

```
vbc /t:exe /optimize+ program_name.vb
ildasm /text /out=program_name_vb.il program_name.exe
```

We expect to get two IL files having comparable semantics, but with slightly differences in the used instructions. The C# and VB.NET programs that we have used as input are shown in Listing 4.2 and Listing 4.3. Because the output of the created IL code is relative large, two excerpts are provided that are used to compare with each other. For the interested reader we disposed the IL code of the metadata and provided the methods in Appendix A.

```
1  class Example {
2    static int theResult;
3    static int Fibonacci(int x) {
4      if (x == 0 || x == 1) {
5         return x;
6      }
7      return Fibonacci(x-1) + Fibonacci(x-2);
8    }
9
10   public static void Main() {
11     theResult = Fibonacci(4);
12   }
13 }
```

Listing 4.2: Fibonacci Example in C#

```
1  Module Example
2    Dim theResult As Integer
3    Function Fibonacci(ByVal x As Integer) As Integer
4      If (x = 0 Or x = 1) Then
5        Return x
6      End If
7      Return Fibonacci(x-1) + Fibonacci(x-2)
8    End Function
9
10   Sub Main()
11     theResult = Fibonacci(4)
12   End Sub
13 End Module
```

Listing 4.3: Fibonacci Example in VB.Net

In this example we use the series of Fibonacci[2]. From the generated IL code, we can see that the body of both the `Fibonacci` instructions are identical to a large extent. There are a few differences that depend on the used compiler. One difference is for example the usage of different comparison strategies for the evaluation part in the if-statement of the body of the `Fibonacci` method. The `if`-statement in line 4 of Listing 4.2 is compiled to the code presented in Listing 4.4, while the `if`-statement in line 4 of Listing 4.3 is compiled to the IL code presented in Listing 4.5. The C# compiler generates IL code that is slightly more optimized than the IL code generated by the VB.NET code.

Although we mentioned that different .NET languages are compilable to IL, we must emphasise that for this research project only C# programs were compiled and disassembled to IL.

[2] The n^{th} number of Fibonacci is calculated according to:

$$Fib(0) = 0$$
$$Fib(1) = 1$$
$$Fib(n) = Fib(n-2) + Fib(n-1), \text{ for } n > 1$$

Thus the series is: $0, 1, 1, 2, 3, 5, 8, ...$

```
10      IL_0000:  ldarg.0              // Load x
11      IL_0001:  brfalse.s  IL_0007   // Return x if x == 0
12
13      IL_0003:  ldarg.0              // Load x
14      IL_0004:  ldc.i4.1             // Load 1
15      IL_0005:  bne.un.s   IL_0009   // Do not return x if x != 1,
16      `                              // otherwise return x
17      IL_0007:  ldarg.0
18      IL_0008:  ret
```

Listing 4.4: IL excerpt obtained from C# compiler

```
11      IL_0000:  ldarg.0              // Load x
12      IL_0001:  ldc.i4.0             // Load 0
13      IL_0002:  ceq                  // Push 1 if x==0, otherwise push 0
14      IL_0004:  ldarg.0              // Load x
15      IL_0005:  ldc.i4.1             // Load 1
16      IL_0006:  ceq                  // Push 1 if x==0, otherwise push 0
17      IL_0008:  or                   // Perform bitwise or on two pushed values
18      IL_0009:  brfalse.s  IL_000d
19
20      IL_000b:  ldarg.0
21      IL_000c:  ret
```

Listing 4.5: IL Excerpt obtained from VB.NET compiler

4.5 Example: IL to ASG

Now that we explained what an ASG is, how it looks like (using a meta-model), and what our translator does, we present an example of translating an IL program to an Abstract Syntax Graph. We have taken the IL program presented in Listing A.3 of Appendix A. Translation of this IL code yields the Abstract Syntax Graph presented in Figure 4.10.

In general cases, we are not interested in the entire ASG but in particular in the simulation results. That is, the executed production rules (presented in the LTS) and the simulation elements of the generated graphs. This is the topic in the next chapter.

4.6 Summary

This chapter started with an introduction of the translator that is implemented to generate an Abstract Syntax Graph from arbitrary IL programs. We have described the different parts (i. e. lexer, parser, and tree walker) of the translator and their purpose. Furthermore, a description of the ASG is presented in the form of meta-models. All nodes and their relationships to other nodes are described.

Also, the design and representation decisions are presented in this chapter. These are decisions with respect to the representation of classnames and namespaces, calculation and representation of method signatures, and representation of identifiers.

We have closed this chapter with a small example of a translation from C# and VB.NET to the Intermediate Language. In this example we show that the used instructions of the resulting IL code are identical to a large extent. Furthermore, we have presented an Abstract Syntax Graph of this IL code to demonstrate how such a graph looks like.

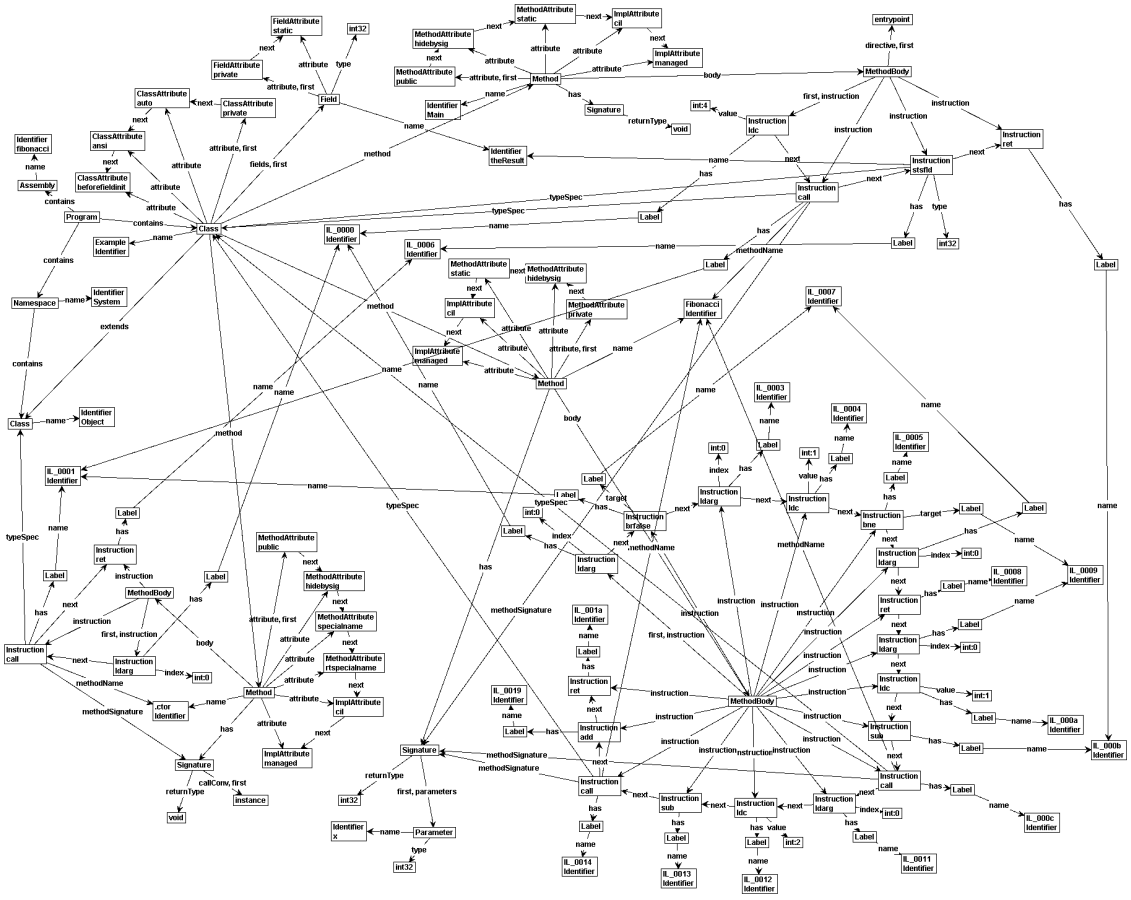


Figure 4.10: ASG of Fibonacci example.

Chapter 5

Specifying IL Semantics with Graph Transformations

This chapter describes how we use graph transformations (explained in Chapter 3) to specify the IL semantics. In order to be able to perform graph transformations, we need a start graph. This graph is generated by the translator described in Chapter 4 and is the Abstract Syntax Graph (ASG) of an arbitrary IL program.

We start this chapter by describing how static analysis is performed by using graph production rules. Furthermore, the ASG implicitly contains control flow information. This control flow information can be made explicit by performing control flow analysis. Then, production rules for control flow analysis should enrich the graph with specific control flow nodes and edges. The decision whether or not to make this control flow information explicit is discussed in Section 5.2.

Section 5.3 introduces what we call the Frame Graph (FG). The FG is used to decorate the start graph with runtime concepts for being able to simulate the execution of a program. In this section we provide a meta-model to describe the FG and discuss implementation alternatives and decisions for some of the concepts present in the Frame Graph. In this section we also describe the Value Graph (VG). The VG describes the relation between the Frame Graph and values.

The production rules describing the semantics for IL instructions are presented in Section 5.4. In general, we use one or two production rules for each instruction. However, this is not always possible because some instructions start a sequence of actions which must be put into effect.

Then, having production rules describing the semantics of a number of IL instructions, it is possible to simulate a program using the GROOVE Simulator. In Section 5.5 we will show this by simulating two example programs.

5.1 Static Analysis

In Chapter 4 we explained that static analysis is partly performed by the translator. There, we have explained that identical string values of identifiers are represented by one single Identifier node. We have chosen to resolve identifiers having identical string values in the translator because comparing two strings by using production rules is more difficult. For example, matching two Identifier nodes (having a pointer to one single node containing a string) is easier than comparing two string nodes for an equal string value. Furthermore, it is hard and costly to use production rules to, for example, determine the signature of a method containing a number of parameters – which can differ per signature – and create a unique node for this. Creating the unique signature node in the translator phase is easier.

Although most of the static analysis is performed in the translator, a (small) part of the static analysis is done with graph transformations. At the moment, this only involves resolving different labels with the same identifier in order to be able to branch to an instruction with the same label

within the same method body. By using a graph production rule the user may get more intuition of what is happening during static analysis.

Labels were already uniquely matched by the string representation of the label names during the translation from IL to graph (see Section 4.3.3). However, in the graph there can be two (or more) labels with the same identifier. In this case, these labels need to be identified, respecting their scope. This means that labels are only resolved to one single label node when the labels are both within the same method body.

The production rule that is used for this is displayed in Figure 5.1.

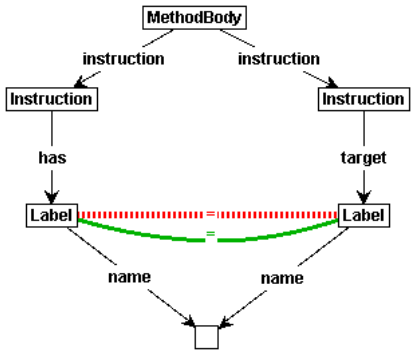


Figure 5.1: Label identification

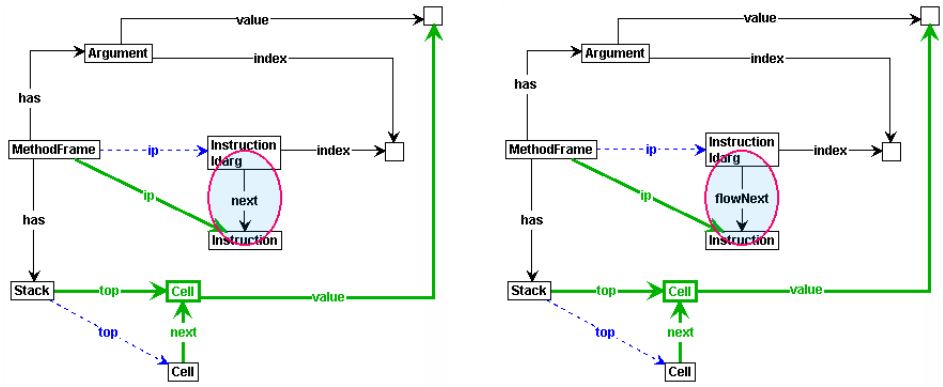
This production rule acts as follows: a matching of this rule can be found if the graph has a MethodBody containing at least two Instructions of which one Instructionhas a Label and the other targets another (i. e. distinct) Label, while both have the same Identifier. If a matching of this rule is found in the graph, the target Label must be merged with the Label of the target instruction. The old target Label and its edge are removed. This way, we represent that target Labels can be resolved to one node, just as it is done during static analysis in a compiler.

5.2 Control Flow Analysis

Because flow control information is implicitly available in the program, there are two possibilities to handle control flow: making it explicit by performing control flow analysis, or using the implicit control flow information during simulation and determine where control should go to after executing an instruction.

Separating control flow from simulation has advantages, such as that the production rules for simulation are becoming slightly smaller. This can make the production rules for both control flow analysis and simulation easier to specify and debug. However, separating control flow from simulation means that for each instruction at least one additional production rule must be defined.

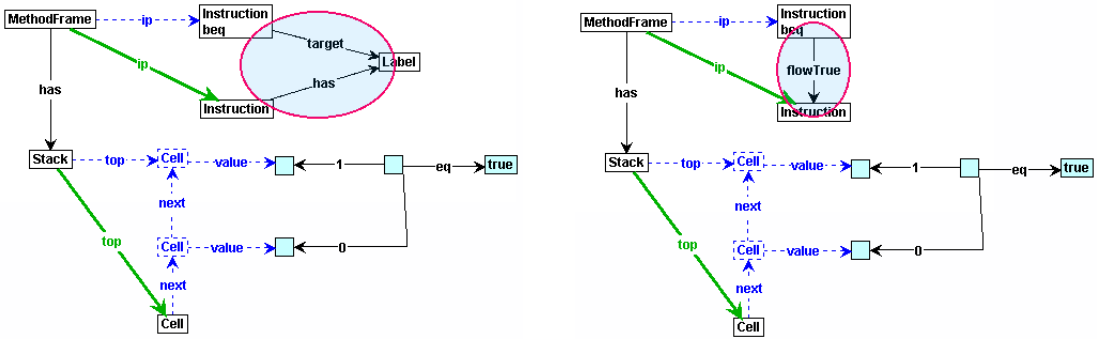
In Figure 5.2 two production rules for the `ldarg` instruction are presented in order to demonstrate the possible differences between having implicit versus explicit control flow information in the graph. This example represents executing an instruction for which the control flow is transferred to syntactically the next instruction, which is the case for the majority of the IL instructions. In Figure 5.2(a) no explicit control flow is available in the graph. If the `ldarg` instruction is executed, the instruction pointer simply needs to be moved to the next instruction which happens to be syntactically the next. When having explicit control flow information – for which an example is presented in Figure 5.2(b) – the rule has to match a flowNext edge in order to know to what node the instruction pointer needs to be moved. In this case, adding the flowNext edge offers no benefit because the semantics of this edge is already represented by the next edge from the ASG. Thus, for these kind of instructions, adding explicit control information involves extra work which does not pay off.



(a) Implicit control flow information (b) Explicit control flow information

Figure 5.2: Implicit versus explicit control flow for the `ldarg` instruction

For an instruction containing non-trivial control flow behaviour, such as the "branch on equal" instruction presented in Figure 5.3, one may expect more gain of having explicit control flow. But when comparing the two rules in the figure, we can see that this is not the case. In Figure 5.3(a) the production rule containing implicit control flow is presented. This rule determines the instruction to branch to on basis of the target label. When making the control flow explicit, the target instruction is determined by performing control flow analysis. Then, the production rule for the `beq` instruction could be as presented in Figure 5.3(b). In our opinion, adding control flow information involves more work – for both specifying the rules and applying the rules to the graph – than simply resolve the target instruction by matching one extra node.



(a) Implicit control flow information (b) Explicit control flow information

Figure 5.3: Implicit versus explicit control flow for the `beq` instruction

Based on the existence of these typical comparisons and the knowledge that most instructions considered in this work are similar to these, we have chosen not to have a separate production system for the generation of a control flow graph because the IL instructions are of such a format that the flow of control can easily be determined at run-time. Besides, we believe that defining additional production rules in order to specify the control flow for each instruction is not worth the extra work. Nevertheless, we are convinced that performing separate control flow analysis provides more insight and intuition.

5.3 Modelling the runtime environment

In order to be able to simulate the run-time behaviour using production rules, the Abstract Syntax Graph will at run-time be enriched with additional elements describing run-time concepts. The Frame Graph contains concepts such as a stack (which is used to store intermediate values) and method frames (holds context information about the methods being executed). We also describe the Value Graph (VG) which represents the objects with their instance fields and data values. The VG also describes the relation between values and arguments, local variables, and cells in the stack. The Frame Graph together with the Value Graph form an Execution Graph (see Section 1.2) representing a run-time state of the system.

This section starts with a formalization and overview of the FG in the form of a meta-model. We also present a meta-model and description of the Value Graph. After that, we describe the decisions and representation of how the stack and method frames are being modelled.

5.3.1 Meta-model of the Frame Graph

Here, we present the meta-model describing the Frame Graph. Each subsection contains a part of the model because presenting the meta-model as a whole would be disorderly.

5.3.1.1 High-level structure

Figure 5.4 contains the overall structure of the FG. Each concept present in the meta-model is discussed by providing a short description.

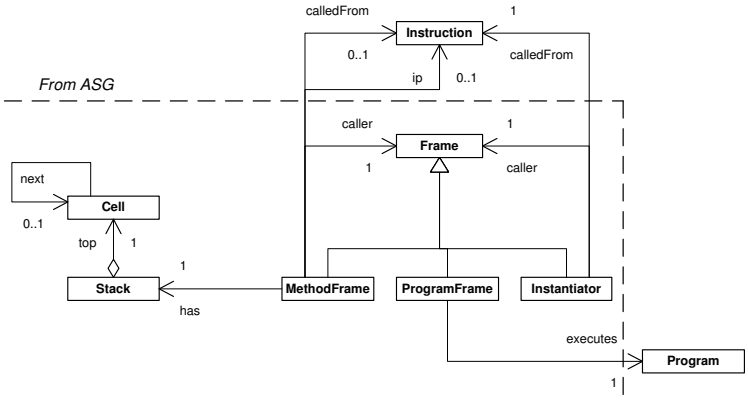


Figure 5.4: Meta-model of the frame graph

The concepts Program and Instruction come from the ASG and are discussed in respectively Section 4.2.1 and Section 4.2.4. The other nodes in the model have the following meaning:

Frame The concept Frame is used for context awareness. There are three different kind of Frames, namely ProgramFrame, MethodFrame, and Instantiator.

ProgramFrame The ProgramFrame holds context information of the program being executed. This Frame is created at the moment the simulation of the program is started and indicates that the program represented by the ASG is being simulated. A ProgramFrame can have different self-loops as represented in Figure 5.5.

These self-loops are used to control the different phases such frames can be in. The meaning of the edges is:

- locateEntrypoint: indicates that the entrypoint (see Section 4.2.1) needs to be located to start execution.

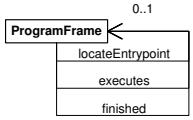


Figure 5.5: Self-loops containing status information

- executes: used to indicate that the program is being executed.
- finished: indicates that program execution has been finished.

Stack The Stack node represents an evaluation stack, which is a run-time concept used to store intermediate values. The Stack contains Cells, of which only the top Cell is referenced. This is indicated by the top pointer. In Section 5.3.3 we discuss the representation of the stack in more detail.

Cell The concept Cell represents a cell of the run-time stack. Each Cell can have a relation (next) to another Cell, making it a stacked representation. Furthermore, a Cell may hold a Value (see Section 5.3.2).

MethodFrame MethodFrame contains information about a method being executed. See Section 5.3.1.2.

Instantiator The Instantiator concept guides the instantiation of new objects (ObjectVal), prior to executing the body of the constructor. See Section 5.3.1.3.

5.3.1.2 MethodFrame

This section provides a description of the MethodFrame and its related concepts.

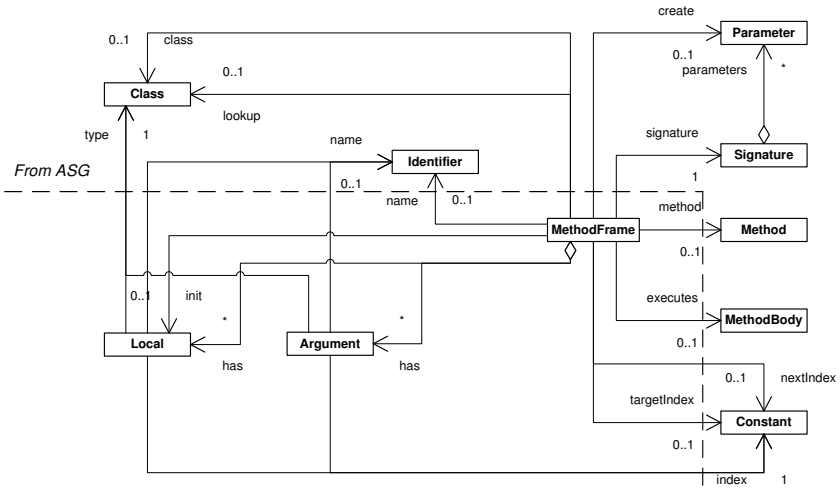


Figure 5.6: Meta-model of the methodFrame

The concepts Method, MethodBody, Signature, Parameter, Identifier, Constant and Class come from the ASG and are discussed in Section 4.2.1.

MethodFrame This node represents a method frame (also known as method state in IL-terminology). A method frame contains context information for a method being executed. As presented in the model of Figure 5.4, MethodFrame has a caller edge to another Frame representing the Frame from which a the MethodFrame is called. MethodFrame can have a calledFrom edge indicating which Instruction caused the creation of this frame. The caller and calledFrom edges are used to transfer control back to the caller Frame and Instruction, when the current frame has finished execution. MethodFrames may have an instruction pointer (ip, see Figure 5.4) pointing to the Instruction to be executed. The MethodFrame does not have a calledFrom edge to an Instruction in case it was created by a ProgramFrame, because the ProgramFrame creates the MethodFrame directly and not by executing an Instruction.

The relations to other nodes depend on the type of call. In case of a `call` instruction, the method frame has a relation with the called Method because this method is already resolved statically (i. e. at compile-time). In case of a `callvirt` instruction, the Method needs to be resolved on the basis of the signature, name, and class of the called method. Performing this method lookup procedure results in a relation of the MethodFramewith the called Method.

In Figure 5.4 we could see that a MethodFrame also has a relation with the Stack to perform its operations on, the caller Frame and the Instruction the method is called from. The MethodFrame always has a relation to the Signature node to be able to determine if there are Arguments that need to be transferred, or to perform a method lookup. The mechanics behind a method frame and transferring arguments is explained in detail in Section 5.3.4 and Section 5.4.3.

Local Local represents a local variable of a Method. A Local has an index, a type, a name. Furthermore, a Local can have a value, which will be further explained in Section 5.3.2.

Argument The Argument concept represents an argument of a method. Argument nodes are created according to the number of Parameter nodes attached to Signature. This process is explained in Section 5.3.4. Argument also has an index, a name in the form of an Identifier, and a type. Also, Arguments are assigned a value (see Section 5.3.2) during the simulation of a method call by production rules. This is further explained in Section 5.3.4.

Constant Although the Constant concept originates from the ASG, it needs some further explanation for its use in the Frame Graph where it represents a constant value used for indexing the Locals and Arguments contained in a MethodFrame. We use the Constant concept in the Frame Graph to make it possible to directly address Locals and Arguments. The edges nextIndexand targetIndexare used to direct the creation of argument nodes – and their unique index value – attached to the MethodFrame node.

The meta-model also contains a create edge to Parameter which is used to create the Argument nodes according to the number of parameters. This process is further explained in Section 5.3.4. The init edge pointing to Local is used to initialize the local variables of a method.

The MethodFrame concept also contains a number of self-edges – similar to ProgramFrame – that are used to control the different phases a MethodFrame can be in. Only one of these edges can be attached to a MethodFrame at the same time. These self-edges are represented in Figure 5.7 and have the following meaning:

- locateArg: The locateArg edge is used to locate if there are any arguments to be transferred. This is done according to the number of parameters in the method's Signature.
- createArgs: If the Signature contains Parameter nodes, then the createArgs edge is created in order to create Argument nodes attached to the MethodFrame.
- transferArgs: When all Argument nodes are created, the actual argument values must be transferred from the Stack. This is accomplished by creating the transferArgs edge.

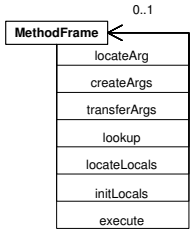


Figure 5.7: Self-loops containing status information

- lookup: The lookup edge is used to indicate that the implementation of a method must be resolved by traversing the class hierarchy.
- locateLocals: locateLocals is used to indicate that the frame has to locate the first local variable (if available).
- initLocals: If locals are available, initLocals indicates that these need to be initialized.
- execute: The execute edge indicates that the method's body is executed.

5.3.1.3 Instantiator

The Instantiator part of the meta-model is responsible for the allocation of new objects and the initialization of its InstanceFields (see Section 5.3.2). The related concepts of Instantiator are shown in Figure 5.8.

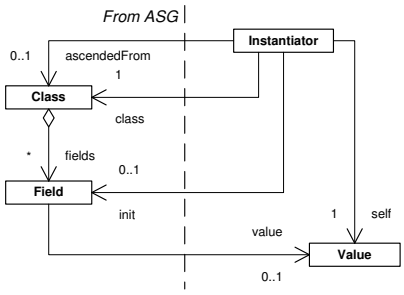


Figure 5.8: Meta-model of the instantiator

The concepts Class and Field come from the ASG and are discussed in Section 4.2.1.

Instantiator The Instantiator concept is used for context awareness and guides the process of instantiating a new object (ObjectVal). The selfedge to the abstract concept Value is during simulation always an edge to ObjectVal, which represents an instance of Class. This is further explained in Section 5.3.2. For convenience, we have added a class-edge to the Class of which ObjectVal is an instance. This edge is used to easily ascend in the class hierarchy to initialize the fields (called InstanceFields) of ObjectVal. The init edge to the Field concept is used during the initialization process of these fields. In case of ascending the class hierarchy, the ascendedFrom-edge indicates from which Class is being ascended and caller (see Figure 5.4) indicates the Instantiator node which calls for ascending. The Instantiator also contains a caller and calledFrom edge (see Figure 5.4) to indicate from which Frame and Instruction the object has been created.

Value The abstract Value concept comes from the Value Graph and can be either a Constant or an ObjectVal. This concept will be explained further in Section 5.3.2.

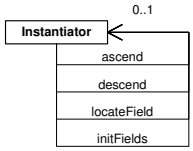


Figure 5.9: Self-loops containing status information

Furthermore, Figure 5.9 shows self-loops – similar to ProgramFrame and MethodFrame – that are used to control the different phases an Instantiator can be in. Such an edge has one of the following labels:

- ascend: During object instantiation, the ascend edge is used to ascend to the top of the class hierarchy in order to locate the fields of a class.
- locateField: The locateField edge is used to indicate that the frame has to locate the first field (if available).
- initFields: If fields are available, initFields indicates that these need to be initialized.
- descend: When no (more) fields are available, descend is used to descend one class in the class hierarchy in order to locate other fields.

5.3.2 Meta-model of the Value Graph

During simulation nodes representing values are created. The sub-graph containing these nodes is called the Value Graph (VG). This section describes the concepts used in the VG by using a meta-model.

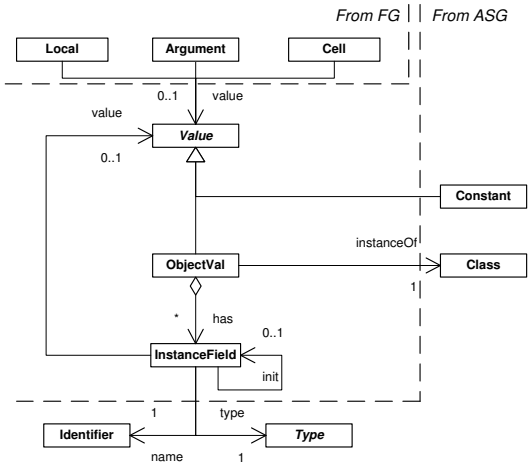


Figure 5.10: Meta-model of the value graph

The concepts Identifier, Type, and Class come from the ASG and are discussed in Section 4.2.1. The concept Constant also comes from the ASG and is discussed in Section 4.2.4. The other nodes in the model have the following meaning:

Value Value is an abstract concept which is used to denote a value, which can be either a Constant or an ObjectVal.

ObjectVal The concept ObjectVal represents an object, which is an instance of a Class. An ObjectVal may contain InstanceFields.

InstanceField An InstanceField represents a field of a class instance (ObjectVal), and has a name and a type. The init self-edge may be used to denote that the InstanceField must be initialized to its default value. Furthermore, an InstanceField may contain a Value, which can be a Constant or an ObjectVal.

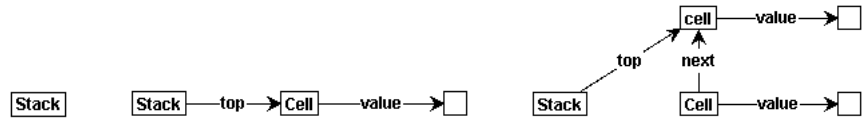
The concepts Local and Argument come from the Frame Graph (Section 5.3.1.2). Cell also is discussed in the Frame Graph (Section 5.3.1.1) and can contain a Value. Note that Cell does not require to contain a Value. This is a design decision and will be further discussed in Section 5.3.3.

5.3.3 Stack

During the execution of a method, an evaluation stack is needed on which instructions can store their (intermediate) values. This section discusses decisions with respect to the representation and implementation of this stack.

5.3.3.1 Representation

Normally, the stack contains a pointer to the top element on the stack. This pointer is increased when new items are pushed on the stack, and decreased when items are popped, i. e. removed from the stack.



(a) Empty stack

(b) Stack containing one element

(c) Stack containing two elements

Figure 5.11: Example stacks containing zero, one and two item(s).

Figure 5.11 contains a possible representation for the straightforward implementation of an evaluation stack. What can be seen here is that we could use a Stack node without a `top`-pointer to represent an empty stack. A Stack node can point to a Cell node that may contain a value (i. e. can have a Value edge to a node) and can have a relation with another Cell node. However, this representation introduces several problems.

The most important problem is that there is no top-pointer available when the stack is empty (Figure 5.11(a)) and there is a top-pointer if at least one item is placed on the stack (Figure 5.11(b)). This means that for every instruction that pushes values on the stack, at least two rules should be needed: one for the case the stack is empty and there is no top-pointer – then the top-pointer and the cell containing a value must be created – and another for the case that the stack is not empty. In the latter case, the top-pointer needs to be moved to the next cell, which a value is assigned to. The same is valid for instructions that pop values from the stack. These kind of instructions need to check whether the value to be popped from the stack is the last value or not. Because if it is, the combination of value, Cell, and the top-pointer must be removed. If it is not the last value, then both the value and the cell must be removed, and the top-pointer must be moved. This means that the number of production rules for simulating instructions that perform actions on the stack are at least doubled.

In the remainder of this section we propose two solutions to this problem and explain our choices.

Alternative I: Top pointer to empty cell One way of solving this problem is that each method frame has its own evaluation stack for which the top-pointer always points to the first empty cell. When a value is pushed to the stack, a new empty cell is created and the top-pointer targets that new cell. When a value is popped from the stack, the value is removed from the secondmost top node, which then becomes empty. After that, the top cell (i. e. the first empty cell) is removed and the top-pointer is positioned to the cell below the old top cell.

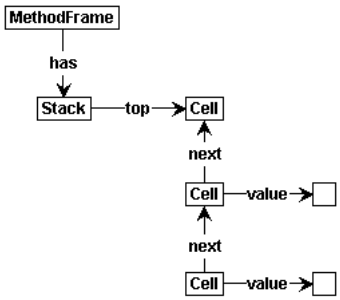
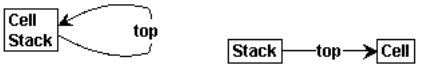


Figure 5.12: Alternative I – Top-pointer points to empty cell.

In this approach, no additional production rules are necessary in order to create a stack of a specific size. However, the production rules for simulating instruction behaviour become a little bit more complex, because they need to add or remove the additional empty cell. The major disadvantage of this approach is that always having an empty cell at top of the stack may be misleading and unintuitive.

Alternative II: Top pointer to cell holding a value Another approach, which resembles the previous alternative, is that each method frame has its own evaluation stack for which the top-pointer points to the last non-empty cell.

A problem now arises when having an empty stack. What should we do with the top-pointer at that moment? In Figure 5.13 we propose two possible solutions for having a stack containing no values. The first alternative is to have the top-pointer point to the stack itself, contradicting with the fact that the top-pointer always should point to a cell. Since this is semantically incorrect, this is not a realistic possibility.



(a) *Top*-pointer to self

(b) *Top*-pointer to empty cell

Figure 5.13: Example stacks containing no items.

The second alternative we discuss is that we let the top-pointer point to an empty cell, and create new cells containing values on top of this empty cell. Although this may not be completely semantically correct, we think this is a reasonable solution. If we apply this approach, the representation of a stack containing two values then would look like represented in Figure 5.14.

The advantage of this representation is that the top-pointer always points to the last non-empty cell, when the stack contains values. Therefore, the production rules use normal stack representations, having a top-pointer to a cell having a value. This is in contrast to the representation proposed in alternative I. A disadvantage of this representation is again, that there is always an empty cell at the bottom of the stack, which is not entirely according the semantics. We prefer this representation of a stack over alternative II because this representation results in more intuitive production rules.

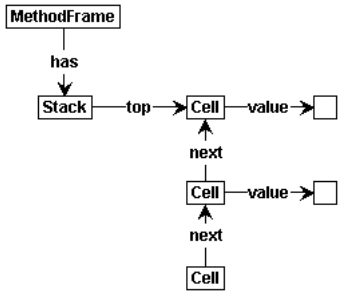


Figure 5.14: Alternative II – Top-pointer points to non-empty cell.

5.3.3.2 Shared Stack

In the previous section we proposed two alternatives for the representation of the stack, for which we preferred the second alternative of having a top-pointer pointing to a non-empty cell in case the stack is not empty. Using this representation of the stack would involve creating a stack (with an empty cell and top-pointer) for each method frame. This has several consequences: arguments need to be transferred from one stack to the other when a method is called, and – for each method call – additional empty cells are created in the graph. Also, when returning from a method, the return value(s) must be transferred back from one stack to the other.

As simplification, we propose to use one single stack for all (nested) methods frames. When sharing a stack, no additional empty cells are created and no additional rules are needed to deal with transferring arguments or return values from one stack to the other. There is only one production rule responsible for creating the stack, namely the production rule for starting execution of the method containing the entrypoint-directive.

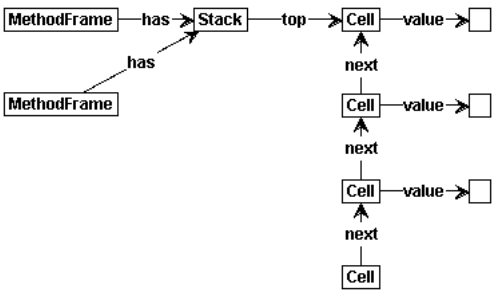


Figure 5.15: Two method frames sharing one stack

We have to mention that when having parallel processes, multiple stacks are needed. In this case, each process should have its own stack. This is not taken into account in this research project.

Based on the advantages described above, we have chosen to use this representation, because then we can keep the production rules as simple as possible, while having few overhead in the graph.

5.3.4 Method Frame Representation and Transferring Arguments

This section discusses the method frame, which is our representation of what in the .NET Framework is called *method state*. Method state is an abstract model that contains information about the environment in which an IL method executes. It consists of information about the method's arguments, local variables, local allocation, and operand stack. The implementation of the method state can be different as long as it preserves its semantics.

In the MS .NET CLR implementation, there is only one continuous stack, containing all this information. If we display this by using a method frame, we get the representation as shown in Figure 5.16. This solution uses one large stack containing sections where the arguments, local variables, and local allocations (omitted in Figure 5.16) are stored. It also contains a section for the operand stack, where the intermediate values of operands and the return value (if any) is stored. The operand stack can grow and shrink when executing instructions. The advantage of this approach is that it gets as close as possible to the actual implementation used in the CLR. A major drawback of this representation is that it is not possible to access the arguments and local variables by its index in one single production rule. This means that additional production rules need to be created to find the argument or local variable corresponding to an index. These production rules must be executed every time we access an argument or local variable. Thus, in a graph transformation setting, this way of addressing the arguments introduces a lot of overhead.

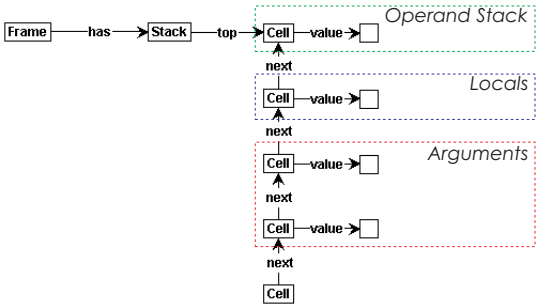


Figure 5.16: Representation with locals and arguments on continuous stack

We choose a representation as presented in Figure 5.17. This solution uses a model of the method state, as proposed by the CLI specification [6]. In this solution, the method frame contains links to arguments and local variables. Each argument and local variable is directly addressable by its index. The disadvantage of this approach is that the values of the arguments need to be transferred from the stack to the method frame. On the other hand, this only needs to be done one single time after each creation of the method frame. After that, both the arguments and local variables are directly addressable by their indices.

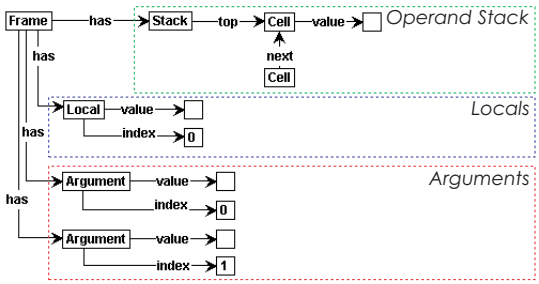


Figure 5.17: Representation with indexed locals and arguments separated from stack

Transferring the values of the arguments from the stack to the method frame, requires more than just one or two production rules, because the arguments are placed on the stack in a low-to-high order. The last argument (say `arg` n) is placed on top. To transfer this last argument and give this argument an index value, we must know how many arguments there are. This however is not known at that moment. So, we first create the argument nodes and their indices. After that, we walk backwards along the arguments while transferring the values from the stack.

This approach will now be explained with an example. Imagine that a method with 3 formal parameters is called. Prior to the method call, the values of the arguments are placed on the stack. We have depicted three integer values 2, 14 and 42 for respectively argument 0, argument 1 and

argument 2. Furthermore, we assume that a frame has been created that knows the signature of the method and has a pointer to the top element of the stack containing the arguments. This is represented in Figure 5.18.

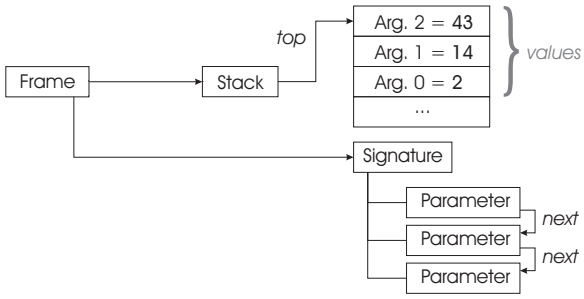


Figure 5.18: Start situation prior to transfer of arguments

The problem is that the values of the arguments must be addressed directly by an index (for example by using the instruction `ldarg 1`). At this moment this is not possible by a simple production rule. In order to make the arguments directly addressable, we need to give them an index. Furthermore, we want to transfer the values of the arguments from the evaluation stack to the MethodFrame node. This is not directly possible, because the last argument is on top of the stack and we do not know the index of the last argument (yet). Therefore, we identify the number of arguments by visiting the parameters described by the signature. The arrows in the figure represent the direction in which the parameters are visited and for which arguments are created. Each time we encounter a Parameter, we create a new Argument node related to the MethodFrame node and give this new node an index. This is represented in Figure 5.19.

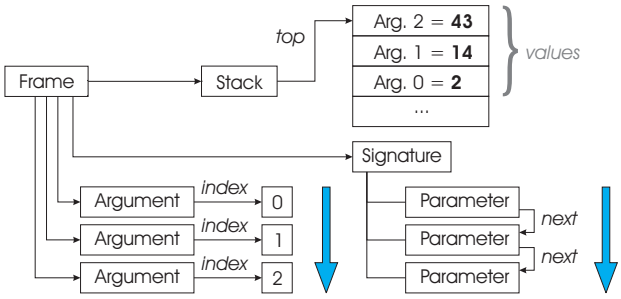


Figure 5.19: Create indexed argument nodes without a value

At the moment that all arguments of the signature have been visited, and thus all arguments are assigned an index value, we can assign values to the arguments. This is done by traversing the arguments of the MethodFrame in a backwards manner and transferring the values from the stack to the arguments. Figure 5.20 represents how this is done. The arrows in the figure represent the direction in which values are removed and assigned. First, the value of the last argument is transferred from the stack to the argument node with index 2. After that, the next argument (with value 14) will be assigned to the argument node with index 1. And finally the last argument value will be assigned to the argument node with index 0.

Now we have achieved that the values of the arguments are directly accessible by addressing them using an index. For example, when using the earlier mentioned instruction `ldarg 1`, a reference will be placed on top of the stack to the value of the argument with index 1. Furthermore, we have omitted information about the local variables in this example. Local variables have a representation similar to the representation of the arguments. However, initializing the local variables works in a different, less complex way.

In our opinion, the solution with having the arguments and local variables separated from the

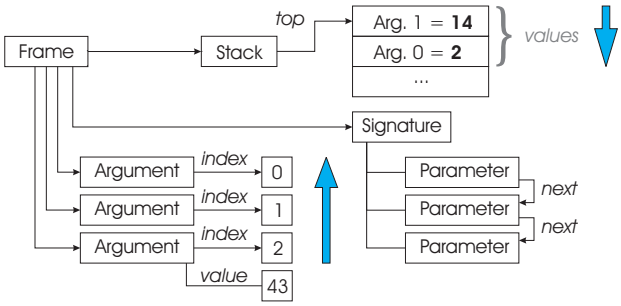


Figure 5.20: Transferring values from stack to argument nodes

stack is more clear than having one continuous stack containing a frame holding the arguments and local variables. In spite of the fact that we need more production rules to transfer the argument values, we think this approach has advantages that justify the use of it for the rest of the production rules, because both the arguments and the stack are now easier to address.

5.4 Production rules

The main goal of this research project is the construction of a set of production rules that describe the semantics of IL instructions in a formal graph-based formalism. In general, we have developed one or two rules for each instruction. However, in some cases this was not doable and additional rules were needed to specify the behaviour. For example, calling a method involves more than just transferring control to the method and executing the method's body: arguments need to be transferred, and in case of dynamic method calls, the method needs to be resolved and local variables must be created and initialized.

This section starts with a description of production rules used for the simulation of starting the execution of a program, creating a new object, and calling a method. After that, a number of common instructions are presented.

5.4.1 Starting Execution

The process of starting the execution of a program is represented in Figure 5.21. The rectangle in this figure represents an intermediate state. The edges represent the application of the rule corresponding to the label of that edge.

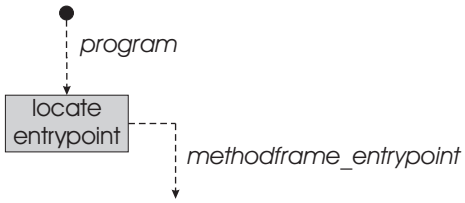
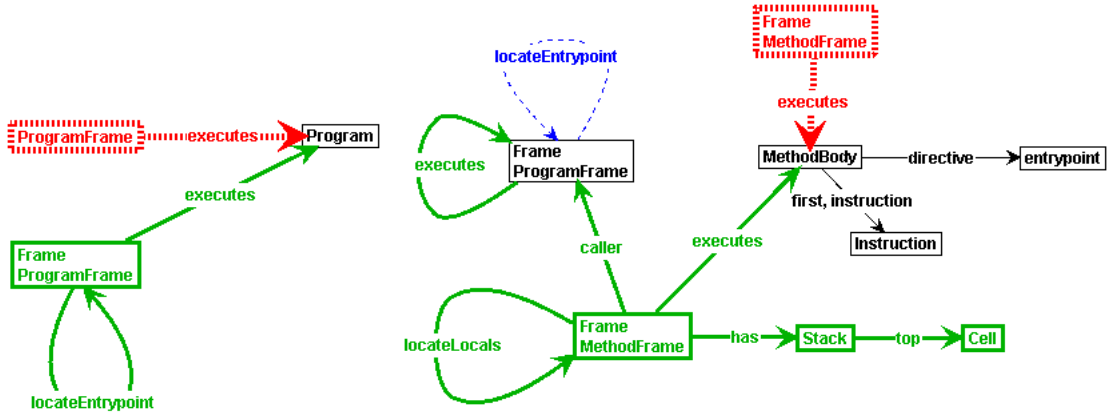


Figure 5.21: Starting program execution with production rules.

As shown in Figure 5.21, the execution of a program starts by applying the `program` rule (see Figure 5.22(a)). This rule creates a ProgramFrame having a locateEntrypoint edge.

Locate Entrypoint When the ProgramFrame containing a locateEntrypoint self-loop is present in the graph, this indicates that the body of the method containing the entrypoint must be located in order to start the execution of that method. This happens only one time during the simulation of a program and is accomplished by the rule `methodframe_entrypoint`. By applying this rule,



(a) program (b) methodframe_entrypoint

Figure 5.22: Starting execution

further execution is achieved by creating a MethodFrame to represent and control the execution of a method. Furthermore, as described in Section 5.3.3, also a Stack with the initial configuration needs to be constructed to perform stack operations on.

5.4.2 Object Creation

The process of object creation is represented in Figure 5.23. Again, as in Figure 5.21, the rectangles in this figure represent intermediate states, while the edges represent the application of a rule.

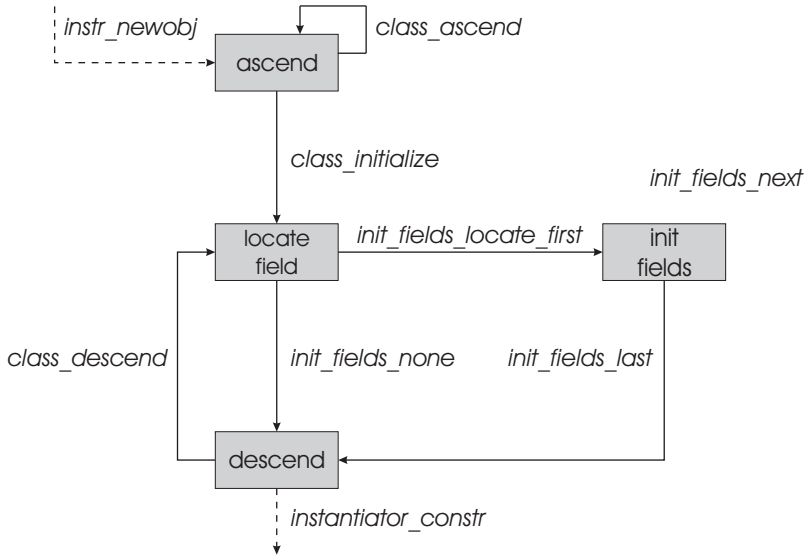
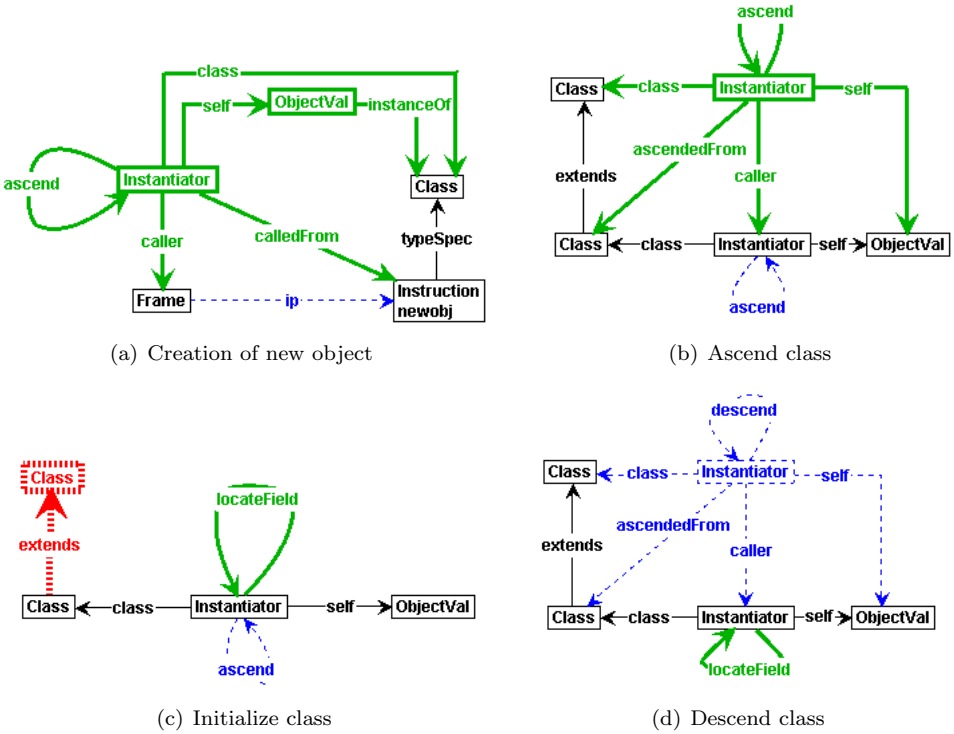


Figure 5.23: Creating a new object with production rules.

Object creation starts with the execution of the `newobj` instruction, which results in the allocation of a new object and execution of the body of the constructor. First, we are going to explain the creation and initialisation of new objects. We start with the execution of the production rule presented in Figure 5.24(a) (`instr_newobj`). This rule ensures that a new instance of the class referenced by the `newobj` Instruction is created. Furthermore, a node called Instantiator is created, which is used to ascend and descend in the class hierarchy, initializing the Fields. The

production rules used for object creation, class initialization, and class ascending and descending, are displayed in Figure 5.24.



(a) Creation of new object

(b) Ascend class

(c) Initialize class

(d) Descend class

Figure 5.24: Object Creation

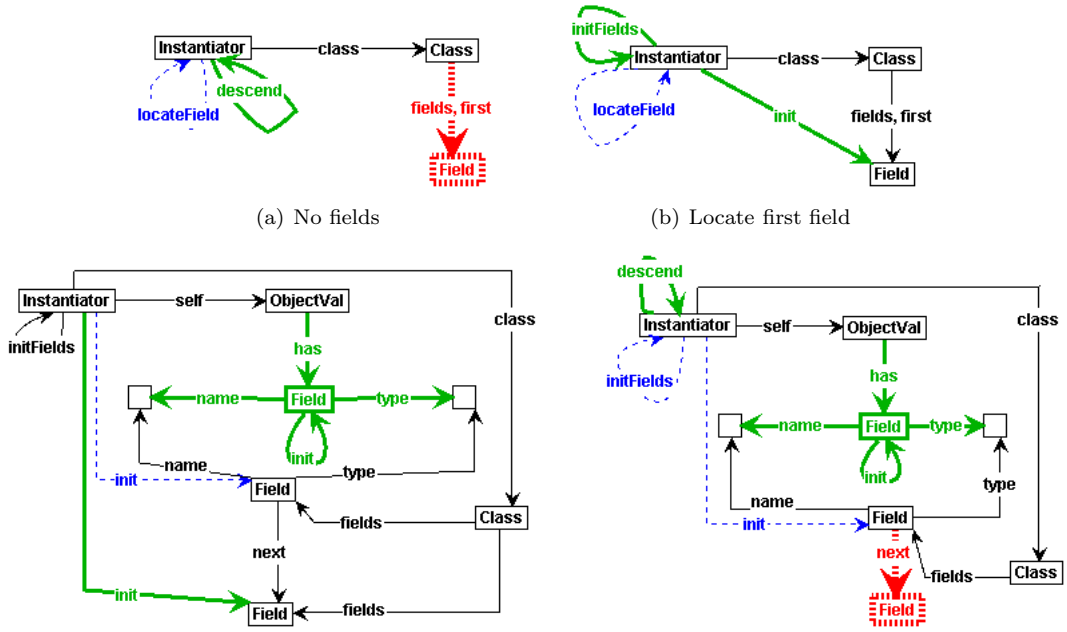
We will now describe the different states and possibilities.

ascend This state is used during the process of ascending the class hierarchy of an object. As long as we are not at the root of the class hierarchy, the rule `class_ascend` (Figure 5.24(b)) is applied. When reaching the top of the class hierarchy the rule `class_initialize` (Figure 5.24(c)) is applied, meaning that we can start with the initialization of fields. Therefore, the first field (if available) must be located.

locate field At the moment a Class contains Fields, these must be initialized. If a Class contains a Field, the rule `init_fields_locate_first` (Figure 5.25(b)) is applied which causes initialization of the available Fields. If a Class does not contain a Field, the rule `init_fields_none` (Figure 5.25(a)) is applied, causing the Instantiator to descend in the class hierarchy or to call the objects constructor.

init fields The initialization of the Fields is processed by another set of rules which is presented in Figure 5.25. The rule shown in Figure 5.25(a) is used when a Class does not have any Fields, and thus no Fields have to be assigned to the object. If this is the case, it is necessary to descend further (if possible) in the class hierarchy. However, when there is a Field available, as displayed in Figure 5.25(b), it is necessary to initialize that Field.

The actual initialization is started by applying the rules presented in Figure 5.25(c) and Figure 5.25(d). The first rule is used when there are other fields that must be initialized and the second rule is used when this is the last field to be initialized. Initializing the fields with a value is accomplished by adding an init edge to that Field and delegate the actual initialization of the Field to another rule (see Appendix C, Figure C.1) having a higher priority.



(a) No fields

(b) Locate first field

(c) Init field and locate next

(d) Init last field

Figure 5.25: Initialization of fields

descend If possible, i. e. there is an ascendedFrom edge, the Instantiator must descend further in the class hierarchy so that other available Fields are initialized. This is accomplished by the rule `class_descend` (Figure 5.25(b)). When the Instantiator returns from ascending the class hierarchy and cannot descend further, all the Fields of the object are initialized and the constructor must be called. The Instantiator cannot descend further when there is no ascendedFrom edge. A lookup for the constructor's method implementation is not necessary, because the .NET compilers automatically provide such an implementation. In order to call the constructor, a MethodFrame is created to which the created object is assigned as the 0^{th} Argument. The rule that is used to accomplish this is `instantiator_constr` and is presented in Figure 5.26.

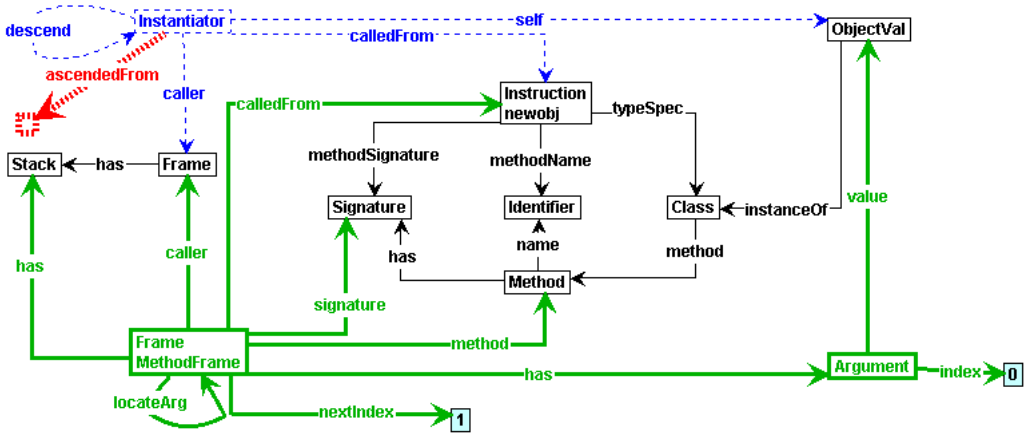


Figure 5.26: Calling the constructor after allocating and initializing an object

Note that a MethodFrame is created. Calling and executing the constructor of an object works according to the same principle as calling a regular method. The rules that are responsible for

this are shown and discussed in Section 5.4.3.

When returning from the `newobj` instruction, i. e. after executing the constructor, a pointer to the new initialized object must be pushed back on the stack. The problem is that this new initialized object is not on the stack of the constructor and thus can not be returned by the `ret` instruction.

Also, because the new initialized object is the 0^{th} argument, connected to the MethodFrame of the constructor, cleaning up this MethodFrame causes the pointer to the new initialized object to be lost. Therefore, we must transfer the pointer to the object before cleaning up the MethodFrame.

It is impossible to solve this problem by simply adding an extra pointer to the newly created object prior to calling the constructor and immediate after allocating the object, because then the pointer to this object may be placed on top of the arguments for the constructor.

The solution we propose is that we have a special rule for the `ret` instruction when the calling instruction is `newobj`. This rule is presented in Figure 5.27 and is responsible for placing the object reference on top of the Stack and removing the MethodFrame.

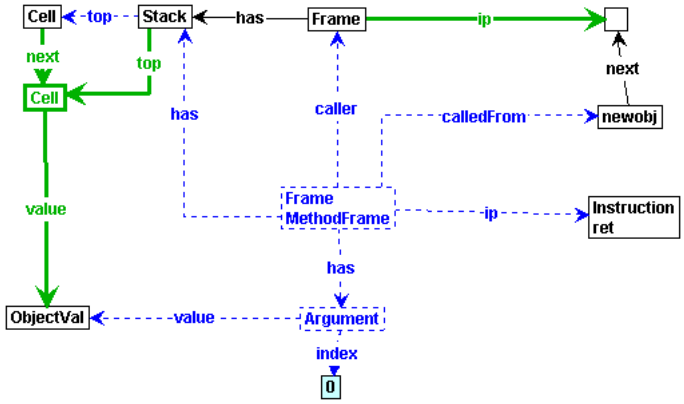


Figure 5.27: Return new object from constructor

5.4.3 Calling methods

In IL, it is possible to have *statically* and *dynamically bound calls*. Statically bound calls use the `call` instruction and the corresponding implementation is bound at compile time. We implemented the production rules for the `call` instruction in such a way that we use the method provided by the call instruction. Dynamically bound calls, on the other hand, use the `callvirt` instruction, The implementation of the method that is called is determined at run-time by performing a lookup starting from the run-time type of the provided object. If the class provides an implementation of a non-static method that matches the method that is searched, the lookup is finished. Otherwise, the lookup process continues searching by checking other base classes in the class-hierarchy.

For both `call` and `callvirt` instructions, the attribute `instance` can be provided. If the attribute `instance` is provided, a pointer to an object must be available on the stack prior to assigning it to the method frame as the 0^{th} argument. In case of a constructor call, the pointer to the newly allocated object is already attached to the method frame by the rules that take care of the allocation and initialization of the new object.

In Figure 5.28 the process of calling different kinds of methods is presented. Each rectangle describes an intermediate state for which one of the different rules is enabled. We describe the different states and possibilities. The graph production rules can be found in Appendix C.

Locate arguments This state describes the point in time, after creating a MethodFrame. That is the moment at which parameters, if available, must be located from the Signature. In this state

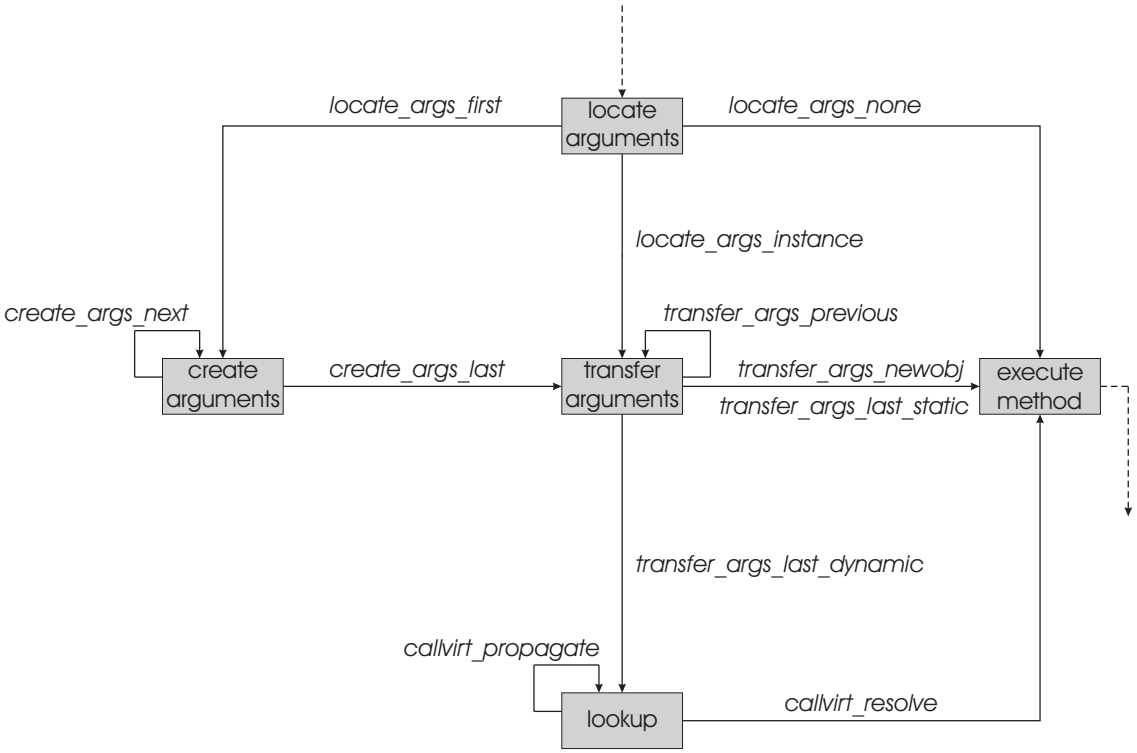


Figure 5.28: Processing a method call with production rules.

there are three possibilities:

1. The Signature of the method contains Parameters, meaning that additional locations for Arguments need to be created at the MethodFrame. This results in the execution of rule `locate_args_first`, and can be the case for both static and object methods (calling convention contains the `instance` attribute).

2. The signature of the method does not contain parameters, but the call is to an object method (`instance` attribute). In this case, the MethodFrame already contains a location for an Argument for the 0^{th} index, but the object needs to be transferred from the stack. The production rule `locate_args_instance` is used for this.

3. The Signature of the method does not contain parameters and the called method is declared static. Then, the method to be called is known and no arguments need to be transferred. The rule `locate_args_none` is executed.

Create arguments At the moment the Signature contains Parameters, the MethodFrame must be prepared by adding empty Argument nodes and edges to them. The rule `create_args_next` creates new Argument nodes until the last Parameter is reached in the Signature, in which case the rule `create_args_last` is executed. Each Argument node that is created has its own index value, meaning that these Argument nodes are directly addressable by the `ldarg` and `starg` instructions. In Section 5.3.4 we have discussed the process of creating and transferring Arguments in more detail.

Transfer arguments When all Argument nodes attached to the MethodFrame are created, the values of the arguments need to be transferred from the Stack to these Argument nodes. The number of arguments to transfer, determine the rule that is executed next: if there is more than

one argument that needs to be transferred from the stack, the rule `transfer_args_previous` is executed. This is repeated until a pointer reaches the 0^{th} Argument attached to the MethodFrame, after which either one of three rules is executed.

1. In case of a constructor call, the 0^{th} argument already contains a value. This value is the object instantiated by the `newobj` instruction. When the 0^{th} argument already contains a value, no arguments are left to be transferred and the `transfer_args_newobj` is applied.

2. When performing a statically bound method call (`call` instruction), the method to be called is already known. If the 0^{th} Argument does not contain a value yet – which is the case for methods calls using the `instance` attribute – the last value for the last Argument must be transferred from the stack. This is accomplished with the `transfer_args_last_static` rule. Immediately after applying this rule, we can proceed with the statically bound method call.

3. If the 0^{th} argument does not contain a value, and the method to be called is not yet known (used instruction for the method is `callvirt`), the `transfer_args_last_dynamic` rule is executed. This rule is responsible for transferring the value of the last argument from the stack to the Argument node. After applying the `transfer_args_last_dynamic` rule a lookup for the implementation of the called method is performed.

Lookup When a dynamically bound call to a method is done, the call to the method needs to be resolved. This is accomplished by matching the name and Signature of the Method to the type of the specified object. If a Method with a matching name and Signature is found in the Class, the rule `callvirt_resolve` is executed. Otherwise, we must ascend the class hierarchy and check again for a matching name and Signature. This is done by the rule `callvirt_propagate`. Notice that looking up Method suffices, because a Method always has a MethodBody containing the implementation of the method.

Execute method This represents the concept of executing a method and is followed by creating local variables and executing the method's body. We have omitted explaining these rules by the figure, because these rules are related with executing a method rather than calling a method. The rules can be found in Appendix C.

5.4.4 Common Instructions

This section discusses a number of common instructions used for branching and arithmetic operations.

Branch Instructions

Branch operations are used to direct control flow. As mentioned before, IL contains conditional and unconditional branch operations. Conditional branch operations only branch when an evaluation yields true, and continue to the next instruction if the evaluation yields false. Unconditional branch operations always branch.

A branch operation always refers to a target, which in IL can be a label or an offset from the beginning of the instruction. If we want to use an offset, the sizes of the other instructions must be known in order to be able to branch to instruction corresponding to the specified offset. Furthermore, in our graph formalism there is no notion of memory, let alone memory addresses, which is why it is impossible to use the offset as a target. Therefore, we are only using labels to branch to. These labels are resolved during static analysis, which means that it is possible to adjust control flow to an instruction with that particular label.

In Figure 5.29 the production rule for an unconditional branch is presented. It is easy to see that the instruction pointer is adjusted to the instruction containing the same label as the target of the instruction `br`, meaning that the control flow is adjusted to the target of the branch operation.

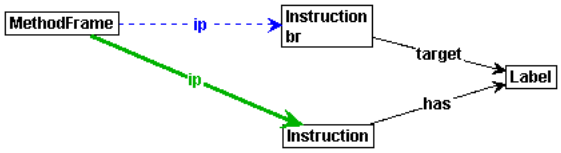
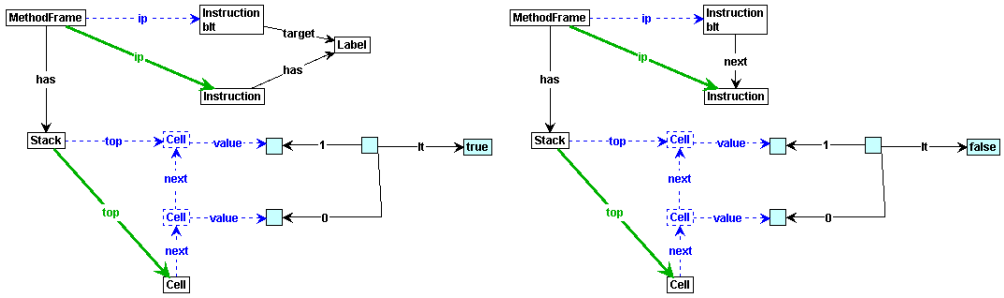


Figure 5.29: Unconditional branch instruction

A conditional branch operation has similar behaviour as an unconditional branch operation but depends on the result of an evaluation. If the evaluation yields `true`, control flow is adjusted to the target of the branch operation. If the evaluation yields `false`, the control flow simply branches to the next instruction. Therefore, two production rules are needed to specify the semantics of the conditional branch. Take, for example, the production rules for the instruction `blt` ("branch on less than") which is presented in Figure 5.30. When assuming that there are two comparable values on the stack and an instruction pointer pointing to the instruction `blt`, one of these rules can be applied. Therefore, the two values from the top of the stack are compared and removed. If the second most top value is less than the topmost value, the comparison yields the boolean value `true` and the rule of Figure 5.30(a) is applicable. This results in having a branch (changing of the instruction pointer) to the instruction having the same label as the target label of the `blt` instruction. If however, this comparison yields `false`, the rule of Figure 5.30(b) is applicable and the instruction pointer is changed to the next instruction.



(a) `blt` yielding `true` (b) `blt` yielding `false`

Figure 5.30: Conditional branch instruction

Other conditional branch instructions (`beq`, `bne`, `bgt`, `brtrue` and `brfalse`) are specified analogously and can be found in Appendix C.

Arithmetic Operations

The implemented arithmetic instructions are `add`, `sub`, `mul`, `div` and `rem`. In Figure 5.31 the production rule for the `add` instructions is presented. The other arithmetic instructions work according to the same principle and can be found in Appendix C. The `add` production rule works as follows. When the instruction pointer points to the `add` instruction, the two top values on the stack are both removed and the result of the operation is placed on the stack.

5.4.5 Limitations

The Intermediate Language contains instructions that perform operations on integers for which an overflow may occur. These instructions contain the term `.ovf` (e. g. `add.ovf`). It is however, impossible to implement these instructions because the GROOVE tool set cannot perform these kinds of operations (yet). For the same reason, we cannot offer support for bitwise instructions (`and`, `or`, `xor`, and `not`).

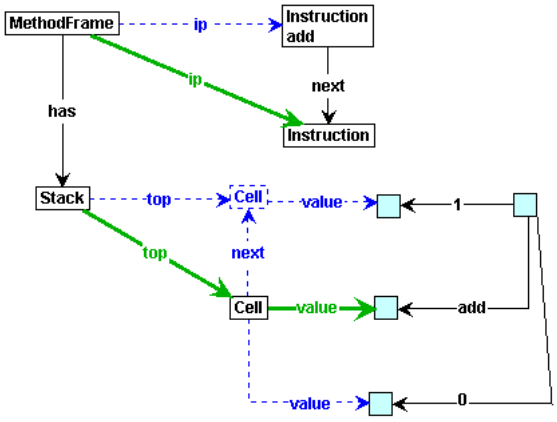


Figure 5.31: Add instruction

5.5 Simulation Examples

This section provides two examples of using an ASG together with the set of production rules in order to simulate the program modelled by the ASG. We do not cover all available production rules during these examples, but we will demonstrate different concepts.

In the first example, the ASG of the Fibonacci IL program (see Section 4.5) is used and simulated by using production rules. This demonstrates the way static calls are processed.

The second example is based on a program of which the building blocks (i. e. the classes) are structured using inheritance. When simulating that program we have to deal with object creation, method resolving, parameter passing, etcetera. For this example a simple C# program is written, compiled and disassembled to IL, which on its turn is translated to an ASG by our translator. This ASG is used as starting point for our simulation. The results of this simulation will be discussed.

Simulating a program yields a Labelled Transition System (LTS), which was already briefly introduced in Section 3.2. A LTS is a graph containing nodes and edges. Each edge stands for the application of a production rule and each node represents a graph. Such a graph can be seen as a state – or snapshot – of the system at run-time.

5.5.1 Example: Fibonacci

For this example, we have taken the ASG of Figure 4.10 as the start graph and applied our production rules to this graph repeatedly. The resulting LTS is presented in Figure 5.32.

According to the IL code (shown in Appendix A, Listing A.3) and the ASG (Section 4.5), executing this program would involve a method call to the `Fibonacci` method, which may result in a number of recursive method calls. The call to the `Fibonacci` method is indicated in the LTS with the dashed rectangle with number 1. An example of a recursive call and returning from that recursive call is indicated with numbers 3 and 4, respectively. The execution of a series of instructions in a method body is indicated with number 5, and returning from the first method call to the `Fibonacci` method is indicated with number 2.

Recall from Section 4.4 that the Fibonacci series looks as follows:

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots, \text{ starting at index } 0$$

According to the IL code, we asked for the 4^{th} number of the Fibonacci series, which is the number 3.

Now, will the end result be as expected? The graph presented in Figure 5.33 shows the final state of the LTS (i. e. the node in the LTS containing the label final). If we look at the highlighted part in this graph, the Field with name `TheResult` contains this value, which indicates that the end result indeed is what we expected.

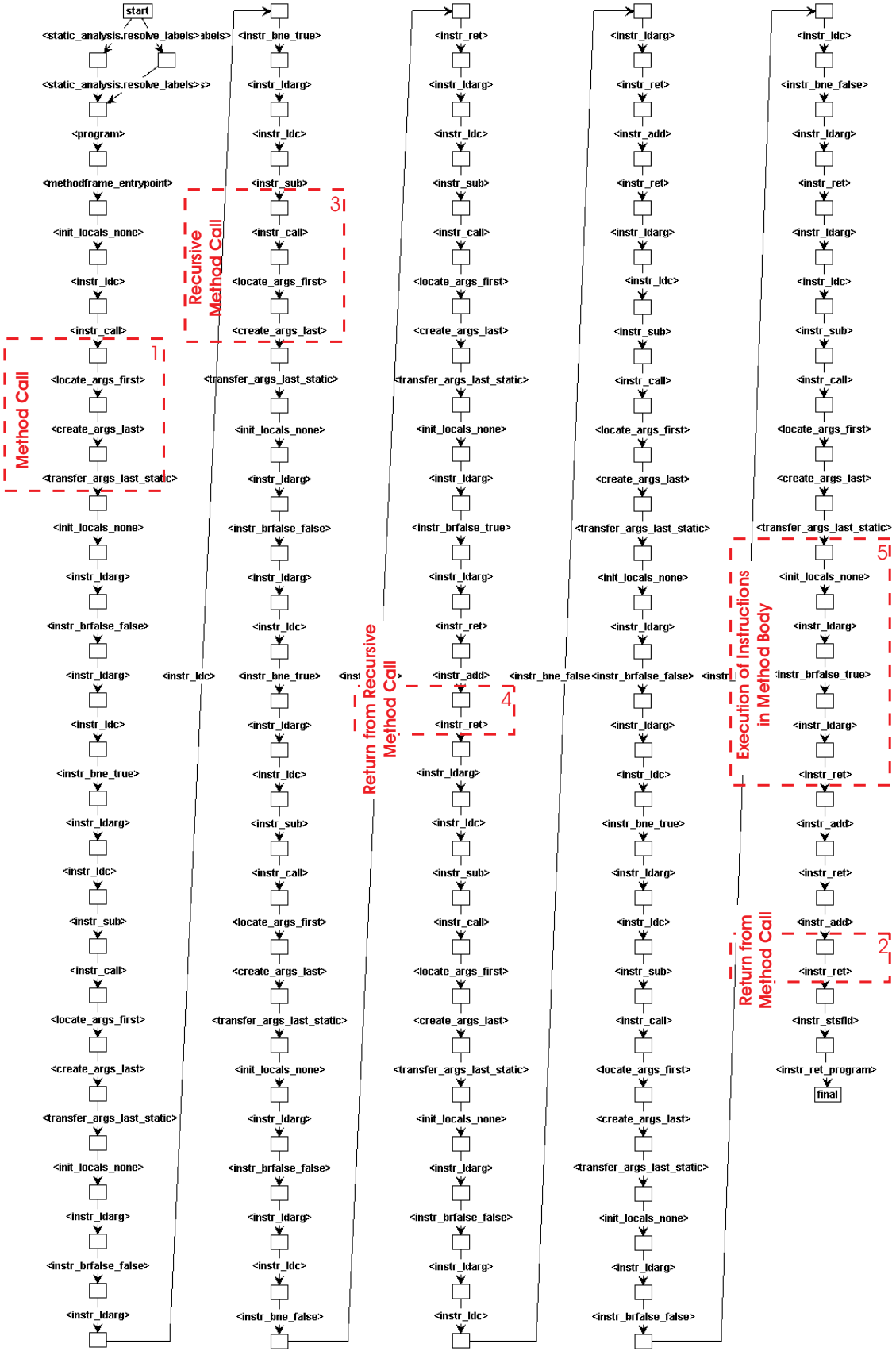


Figure 5.32: LTS of the Fibonacci Example

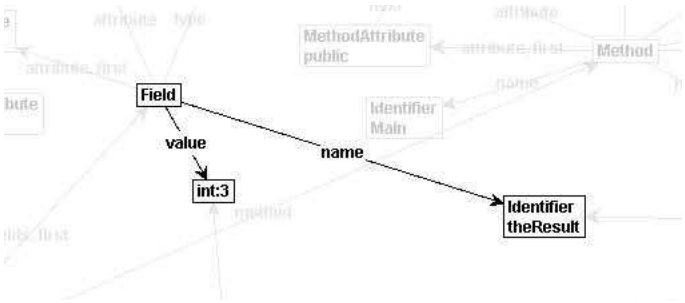


Figure 5.33: Result of the Fibonacci Example

5.5.2 Example: Calculator

In this example we demonstrate a simple calculator program that has to deal with object creation, method-lookups, method overriding, and parameter passing. The class-diagram in Figure 5.34 shows the structure of the program. In this model we show that three different classes (`CalcAdd`, `CalcDiv`, and `CalcMean`) inherit from the base class `Calc`. The base class contains a virtual (i. e. overwritable) method `apply`, which provides a default implementation. Class `CalcAdd` does not overwrite this method, but the classes `CalcDiv` and `CalcMean` do.

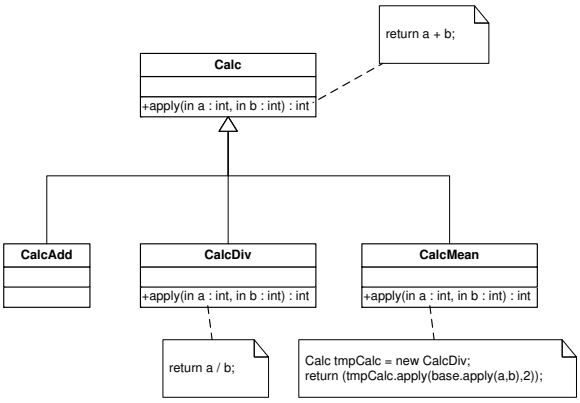


Figure 5.34: UML model of the Calculator Example

The C# program displayed in Listing 5.1 is an implementation based on the presented model. In the `main` method of class `Test` three instances are created, one of each child-class. For each instance, we call the `apply` method and store the return value of the method in a global field.

Compiling and disassembling this C# program yields IL code, which is translated to an ASG. Both the IL code and the ASG are not presented here, but the interested reader can find them in Appendix B.

The ASG is used as start graph for our production system. Applying the production rules yields the LTS presented in Figure 5.35.

In this LTS a number of things are worth mentioning. One of these things is object creation. From the code, the `Main` method instantiates three objects (`CalcAdd`, `CalcDiv`, and `CalcMean`) and the `apply` method in class `CalcMean` also creates an object of type `CalcDiv`. Object creation is indicated in the LTS by the execution of the `instr_newobj` rule. For example, see the dashed rectangles numbered 1 and 8 (the first rule) which represents the creation of the objects of `CalcAdd` and `CalcDiv`, respectively. The object creation of number 1 is followed by the object instantiation (2), which involves ascending and descending in the class hierarchy in order to initialize the fields of the parent classes, if available. After object instantiation is finished, the constructor of the instantiated object is executed (indicated with 3). During execution of the constructor of the class

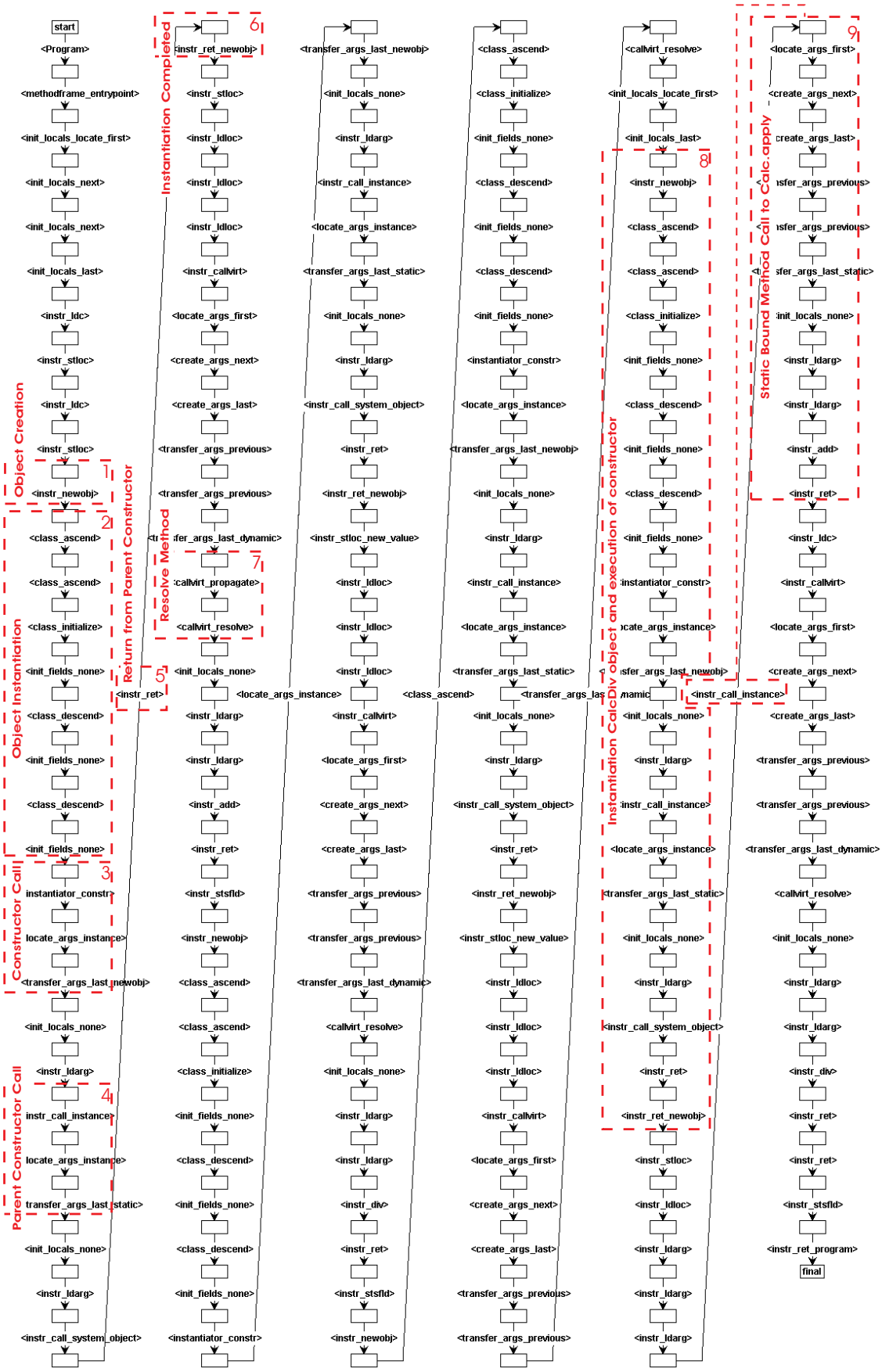


Figure 5.35: LTS of the Calculator Example

```
class Calc {
  public virtual int apply(int a, int b) {
    return a + b;
  }
}

class CalcAdd : Calc {
}

class CalcDiv : Calc {
  public override int apply(int a, int b) {
    return a / b;
  }
}


class CalcMean : Calc {
  public override int apply(int a, int b) {
    Calc tmpCalc = new CalcDiv();
    return (tmpCalc.apply(base.apply(a,b),2));
  }
}

class Test {
  private static int x = 0;
  private static int y = 0;
  private static int z = 0;

  public static void Main() {
    int a = 10;
    int b = 2;

    Calc calc = new CalcAdd();
    x=calc.apply(a, b);
    // x == 12

    calc = new CalcDiv();
    y=calc.apply(a, b);
    // y == 5

    calc = new CalcMean();
    z=calc.apply(a, b);
    // z == 6;
  }
}
```

Listing 5.1: Calculator Example in C#

`CalcAdd`, the constructor of its parent-class is called. This is indicated with number 4. After the parent' s constructor has been executed, control flow is transferred to the constructor of `CalcAdd` (5). Immediately after the return from the parent-class, the constructor of `CalcAdd` has finished executing and returns (6), leaving the new instantiated object of type `CalcAdd` on the stack. From this object, the method `apply` is called, which propagates the method call to its base class. This is displayed by the rules executed in the rectangle with the number 7.

As mentioned earlier, number 8 also indicates the creation of an object. In fact, it represents the allocation of the `tmpCalc` object – of type `CalcDiv` – as part of the method body of the `apply` method of class `CalcMean`. The rectangle with number 8 visualizes the initialization of an object (i. e. the resolving of fields) and that the execution of the constructor. The last production rule represents the return instruction at the end of the constructor, which causes the new created object to be left on the stack. The execution of the production rules indicated by 9 represent a method call to the `apply` method of the class `CalcDiv`. This method is statically bounded (i. e. `call` instruction) and overrides the virtual `apply` method of class `Calc`.

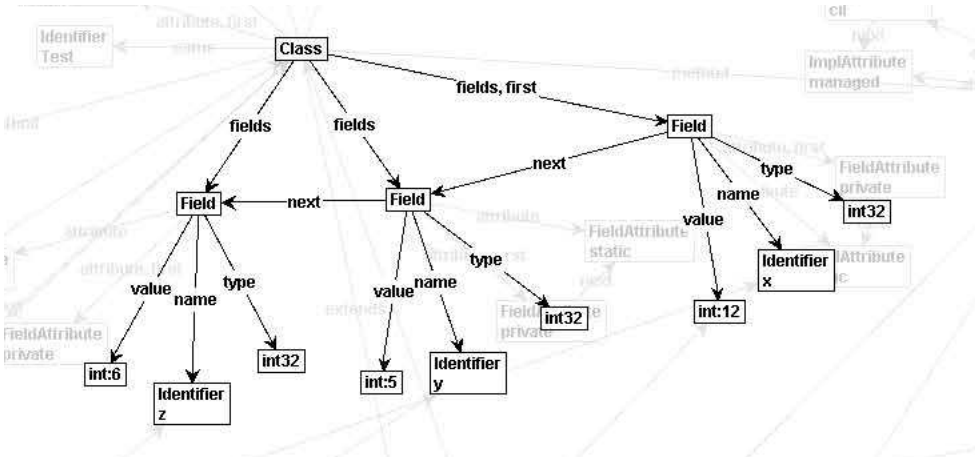


Figure 5.36: Final Graph of the Calculator Example

Now that we have seen which production rules are applied, we want to make sure that the result of our simulation is the same as expected. That is, the fields `x`, `y`, and `z` should yield 12, 5, and 6, respectively. In contrast to the previous example, we present only the part of the final graph containing these fields and their values in Figure 5.36. Again, we can see that the values indeed correspond with what we expected.

5.6 Performance

Although we have not performed any explicit performance testing during this research project, a few words must be said about our experiences during simulation of the examples. In the program used for the Fibonacci example we requested the 4^{th} number of the Fibonacci series. Simulating this example with a linear exploration strategy was not a problem for the simulator, which came up with a LTS consisting of 136 states and 135 transitions within a few seconds. When calculating the 8^{th} number of the series, the LTS consists of 973 states and 972 transitions. However, when trying to look at the graph representing the last state, a list of errors with respect to the layout algorithm appeared. Thus, it was impossible to check whether or not the simulation yields the correct result.

When performing the same test (i. e. calculating the 8^{th} number of the Fibonacci series) with the generator tool, we obtain the following output:

```
Statistics: States: 972
  Explored: 1
  Transitions: 971

Time (ms): 19485
  Matching: 610  3.1%
  Transforming: 3718 19.1%
  Iso checking: 14986 76.9%
  Building GTS: 77  0.4%
  Measuring: 62  0.3%
Space (kB): 270513
```

From this output we can read the number of yielded states and transitions. It also contains the distribution of the time over the different performed tasks during the simulation.

However, when trying to have the generator output the LTS to a file, it crashes with a null-pointer exception and the resulting LTS was never written. This means that we were still not

able to determine if this simulation was performed correctly. Furthermore, the fact that we have 1 state and 1 transition less than we had in the simulator is confusing. The exact reason for this is unknown.

Simulating the Calculator example by using the simulator did not cause any problems and resulted in a LTS consisting of 184 states and 183 edges within a few seconds. When performing the same test with the generator, this yields the following output:

```
Statistics: States: 184
  Explored: 1
  Transitions: 183

Time (ms): 2640
  Matching: 45  1.7%
  Transforming: 1938 73.4%
  Iso checking: 626 23.7%
  Building GTS: 16  0.6%
  Measuring: 0  0.0%
Space (kB): 68731
```

Trying to output the LTS to a file yields the same problem as described for the Fibonacci example.

5.7 Summary

Although static analysis is partly performed by our translator, we have developed one production rule that is responsible for static analysis with production rules. This production rule involves resolving target Labels to the Label of an Instruction.

Furthermore, we have discussed whether or not to perform flow graph analysis, and concluded that introducing a separate flow graph analysis phase does not pay off. The reason for this is that the syntactic order of appearance of instructions is in most cases equal to the semantic order of execution. Performing separate flow graph analysis would involve specifying extra production rules, while there is almost no added value.

A meta-model of the Frame Graph (FG) is presented and discussed in Section 5.3.1. The FG is the ASG extended with run-time information, such as a stack and method frames. We also provided a meta-model of the Value Graph (VG) which describes values and their relation to the FG and ASG. This is discussed in Section 5.3.2.

We have discussed decisions about modelling the runtime environment in Section 5.3. This involved an approach of representing the stack and the method state, but also discusses transferring arguments from stack to method frame.

Finally, after discussing implementation decisions, we presented and explained a number of the developed production rules in Section 5.4. We concluded this chapter by simulating two example programs. This is accomplished by applying a set of production rules to the ASG of each program, resulting in a Labelled Transition System (LTS). Such a LTS represents the (intermediate) graphs as nodes and applied production rules as edges between these nodes. For these two examples, we have provided a few comments on the performance of the GROOVE tool set.

Chapter 6

Conclusion

Using graphs and graph transformations to specify the IL semantics offers advantages. First, representing the semantics in a graphical way is both intuitive and non-ambiguous. And second, the formal background of graph transformations opens the possibility to use formal verification techniques.

In this project we aimed at a graph-based representation of the semantics of the .NET Intermediate Language. This resulted in a tool and a set of production rules describing IL semantics. The tool is a translator that is able to transform an arbitrary IL program into an Abstract Syntax Graph, which is used as start graph to which the production rules are applied. By applying the production rules we can simulate the behaviour of the original IL program.

The graph transformations that we developed describe the semantics of the IL instructions, meaning that each instruction needs to be specified only once. Changes to an IL program only influences the start graph (ASG), but not the production rules themselves. The production rules that we have constructed cover techniques such as object creation, method calling, and inheritance. Furthermore, we have constructed production rules that are able to perform arithmetical operations, branching operations, and load and store operations.

At the end of this project we have presented the simulation of two example programs containing different (object-oriented) language concepts. The applied production rules are presented in the generated Labelled Transition Systems of these simulations. Furthermore, we have shown that simulating the programs yields correct results.

We have demonstrated some encouraging results and believe that we have provided a solid base for future research on using graph transformations to specify the semantics of the .NET Intermediate Language, and possibly other languages too.

6.1 Discussion

In this section we will evaluate choices and decisions with respect to our implementation. Furthermore, we will briefly discuss our experiences with the GROOVE tool set and give our opinion about the used approach.

6.1.1 Implementation

Method signatures are dealt with in the translator; the translator is responsible for matching and merging identical signatures. An alternative of doing this in the translator would be doing it by using graph production rules. However, using graph production rules for signature creation and signature matching is difficult. This is because determining the method signature would involve determining the parameters (and their order) of a method. After determining the method's signature, a check must be performed if this signature does not already exist. If so, the two signatures must be merged to one single (and unique) signature. The last step of comparing and

merging two signatures with production rules may prove to be very difficult, because comparing structures is quite hard.

Furthermore, we have mentioned that when a `call` instruction is used, we call the method provided by that instruction. This is not entirely according to the IL semantics. Normally a lookup to the implementation of the called method should be performed by traversing the class-hierarchy, but in our rules we assume that the implementation of the called method can always be found at the destination provided with the `call` instruction. We do this because we are almost certain that IL programs obtained from the C# compiler always refer to the implementation of a method in case of a `call` instruction.

6.1.2 GROOVE

During this project we have intensively worked with the editor, simulator and imager of the GROOVE tool set. Installing the tool set is easy, and working with the tools is very intuitive.

While GROOVE provides us a nice set of tools, the development of these tools went on during our project. Although this resulted in a quick response to bugs that were found, it also resulted in the introduction of new bugs. Due to this, every now and then we had to determine whether or not a problem concerned our production rules, or if it was a mere bug in the GROOVE tool set. Sometimes, this was very time consuming.

6.1.3 Approach

Designing and implementing the translator proved to be more difficult than expected. For creating the translator with ANTLR we used an existing parser grammar file[1] that turned out to contain some non-deterministic rules. These had to be corrected. With hindsight, we must say that it would have been better to determine a subset of the Intermediate Language and write a translator for this subset only. This would have considerably decreased the complexity of the language to be translated.

With respect to the approach of using graphs and graph transformations in order to specify the semantics of a dynamic language, we would like to state that we believe that using this approach has proved to be working. We were able to specify the operational semantics of a number of IL instructions by using graph production rules. Furthermore, we were able to represent run-time state snapshots by using a graph, while transitions between two graphs – which are obtained by applying production rules – represent run-time changes during simulation. Although graphs and graph transformations are very useful for representing the semantics, we think it also has some shortcomings. One of these is that it is (at this moment) impossible to implement instructions that depend on memory locations (for example loading the address of a local variable). Another shortcoming is that sometimes a trick had to be applied in order to get something working. An example of such a trick is our used representation of the stack.

6.2 Related Work

Work that is related to ours is by Corradini et al.[4]. They use graph transformations to formalize a large fragment of Java. In their proposal, one rule is generated for each method, making the rules dependent on the program. Our work is more general than theirs, because we are able to simulate the behaviour of the individual instructions, instead of just the result of the execution of a method.

Kastenberg et al.[13] present operational semantics of a self-defined imperative, object-oriented language (called TAAL). They use graph transformation rules to extend a flat abstract syntax graph with control flow information, which again is used as the start graph for simulation. In our solution, we do not extend our abstract syntax graph with explicit control flow information, but instead use the implicit control flow information that is already available.

[1] The grammar file was written by Pascal Lacroix and can be obtained from `http://www.antlr.org`

Arends[3] presents work that involved the implementation of a translator that produces graph production rules from Java bytecode. Templates with variable labels are used to build a production rule for each situation by inserting the right labels. This differs from our approach, because we use generic production rules which do not have to be generated according to the executed program.

6.3 Future Work

Both the translator and the created production rules need further extension. At this moment we are only able to process single-file assemblies. This should be expanded to multi-file assemblies. Also, not implemented are the concepts that deal with – among others things – exception handling, threads, boxing and unboxing, generics, and type conversion. Furthermore, bitwise instructions, the switch statement, and instructions directly addressing memory locations are not implemented. This is all left for future work.

It may be interesting to research if it is possible to execute graph transformation rules controlled by some other means, for example the CLR. The start graph then does not have to represent the program to be simulated (as in case of the ASG), but only a few runtime concepts such as a stack. Simulation then may be achieved by executing a program step by step and call a production rule for each instruction, rather than first create an Abstract Syntax Graph and perform graph transformations to this ASG. If this works, translating an IL program is not necessary any more and it is sufficient to just have a set of graph production rules describing the semantics of the IL instructions.

With respect to GROOVE we must mention that not all of the instructions are represented by graph transformation rules because the GROOVE tool set does not provide support for all kinds of operations and types. For example, GROOVE does not support the bitwise operations, floating point types and unsigned integers. To support these instructions, adjustments to the GROOVE tool set are necessary.

What we miss in the simulator and editor is a good layouting algorithm. Especially when graphs are becoming large, layouting this graph – which mostly must be performed by hand – is a time-consuming and unpleasant process. Furthermore, when applying a production rule to a graph with hidden nodes and edges, the whole graph becomes visible again. It would be very helpful if only the part of the graph that was not hidden would stay visible, along with the newly added nodes and edges. It might also be good to give the user the possibility of colouring nodes and edges in a graph, so that they can be quickly found during simulation. Another suggestion is to introduce some way of grouping nodes and edges in order to make the graph more orderly. Also, the larger a graph gets, the slower the user interface is going to work. We suggest that this is a problem that should be dealt with.

Furthermore, we would like to emphasise that we have mainly used the simulator in order to test and debug our production rules, which proved to be very helpful. However, as mentioned in Section 5.6, we accounted problems with layouting a large LTS. Furthermore, the generator was not able to export the LTS to a file. In our opinion, these problems need to be solved.

Bibliography

[1] L. Aceto, W. Fokkink, and C. Verhoef. Structural operational semantics, 1999.

[2] ANTLR website. `http://www.antlr.org/`.

[3] Mark Arends. A simulation of the java virtual machine using graph grammars. Master's thesis, University of Twente, November 2003.

[4] Andrea Corradini, Fernando Luís Dotti, Luciana Foss, and Leila Ribeiro. Translating java code to graph transformation systems. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *ICGT*, volume 3256 of *Lecture Notes in Computer Science*, pages 383–398. Springer, 2004.

[5] Joe Duffy. *Professional .NET Framework 2.0.* Wiley Publishing, Inc., April 2006.

[6] ECMA International. Ecma international, Common Language Infrastructure (CLI), Standard ECMA-335. *http://www.ecma-international.org/publications/standards/Ecma-335.htm*, June 2005.

[7] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation.* Springer-Verlag, 2006.

[8] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation - part ii: Single pushout approach and comparison with double pushout approach. In Rozenberg [25], pages 247–312.

[9] GROOVE website. `http://groove.sourceforge.net/`.

[10] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundam. Inform.*, 26(3/4):287–313, 1996.

[11] Java technology website. `http://java.sun.com/`.

[12] Harmen Kastenberg, Anneke Kleppe, and Arend Rensink. Defining object-oriented execution semantics using graph transformations. In R. Gorrieri and H. Wehrheim, editors, *Proceedings of the 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *Lecture Notes in Computer Science*, pages 186–201. Springer-Verlag, 2006.

[13] Harmen Kastenberg, Anneke Kleppe, and Arend Rensink. Engineering object-oriented semantics using graph transformations. CTIT Technical Report 06-12, University of Twente, March 2006.

[14] Harmen Kastenberg and Arend Rensink. Model checking dynamic states in GROOVE. In A. Valmari, editor, *Model Checking Software (SPIN)*, volume 3925 of *Lecture Notes in Computer Science*, pages 299–305. Springer-Verlag, April 2006.

[15] Serge Lidin. *Inside Microsoft .NET IL Assembler.* Microsoft Press, Redmond, WA, USA, 2002.

[16] Microsoft Corporation. Overview of the .net framework. Retrieved June 28, 2006, from `http://msdn2.microsoft.com/en-us/library/a4t23ktk.aspx`, 2003.

[17] Microsoft Corporation. Common language specification. Retrieved December 24, 2006, from `http://msdn2.microsoft.com/en-us/library/12a7a7h3.aspx`, 2006.

[18] Microsoft Corporation. Microsoft Portable Executable and Common Object File Format Specification. Retrieved July 4, 2006, from `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`, May 2006.

[19] Hanspeter Mössenböck, Wolfgang Beer, Dietrich Birngruber, and Albrecht Woess. *.NET Application Development: With C#, ASP.NET, ADO.NET, and Web Services*. Pearson Addison Wesley, 2004.

[20] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[21] Arend Rensink. The GROOVE simulator: A tool for state space generation. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *AGTIVE*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer, 2003.

[22] Arend Rensink and Harmen Kastenberg. GRaphs for Object-Oriented VErification (groove). Retrieved Apr 1, 2006, from `http://groove.sourceforge.net`, April 2006.

[23] Jeffrey Richter. Garbage collection: Automatic memory management in the microsoft .net framework. *MSDN Magazine*, November 2000.

[24] Jeffrey Richter. *Applied Microsoft .NET Framework Programming*. Microsoft Press, Redmond, WA, USA, 2002.

[25] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[26] Thuan Thai and Hoang Q. Lam. *.NET Framework Essentials*. O'Reilly, 2nd edition, February 2002.

[27] G. Winskel. *The formal semantics of programming languages*. The MIT Press, 1993.

Appendices

Appendix A

IL programs side to side

This appendix contains the resulting IL code after compiling two identical programs to the Intermediate Language. The two input programs are written in the .NET languages C# and VB.NET and are presented in Listing A.1 and Listing A.2.

Compiling (and disassembling) these two programs yields two files containing IL code. We only present the most interesting part of the IL files. That is, metadata and other superfluous methods are omitted. The resulting IL code can be found in Listing A.3 and Listing A.4.

```
class Example {
  static int theResult;
  static int Fibonacci(int x) {
    if (x == 0 || x == 1) {
       return x;
    }
    return Fibonacci(x-1) + Fibonacci(x-2);
  }

  public static void Main() {
    theResult = Fibonacci(4);
  }
}
```

Listing A.1: Fibonacci Example in C#

```
Module Example
  Dim theResult As Integer
  Function Fibonacci(ByVal x As Integer) As Integer
    If (x = 0 Or x = 1) Then
      Return x
    End If
    Return Fibonacci(x-1) + Fibonacci(x-2)
  End Function


  Sub Main()
    theResult = Fibonacci(4)
  End Sub
End Module
```

Listing A.2: Fibonacci Example in VB.Net

```
.class private auto ansi beforefieldinit Example
       extends [mscorlib]System.Object
{
  .field private static int32 theResult
  .method private hidebysig static int32
          Fibonacci(int32 x) cil managed
  {
    // Code size       27 (0x1b)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  brfalse.s  IL_0007

    IL_0003:  ldarg.0
    IL_0004:  ldc.i4.1
    IL_0005:  bne.un.s   IL_0009

    IL_0007:  ldarg.0
    IL_0008:  ret

    IL_0009:  ldarg.0
    IL_000a:  ldc.i4.1
    IL_000b:  sub
    IL_000c:  call       int32 Example::Fibonacci(int32)
    IL_0011:  ldarg.0
    IL_0012:  ldc.i4.2
    IL_0013:  sub
    IL_0014:  call       int32 Example::Fibonacci(int32)
    IL_0019:  add
    IL_001a:  ret
  } // end of method Example::Fibonacci

  .method public hidebysig static void  Main() cil managed
  {
    .entrypoint
    // Code size       12 (0xc)
    .maxstack  8
    IL_0000:  ldc.i4.4
    IL_0001:  call       int32 Example::Fibonacci(int32)
    IL_0006:  stsfld     int32 Example::theResult
    IL_000b:  ret
  } // end of method Example::Main

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method Example::.ctor

} // end of class Example
```

Listing A.3: Fibonacci C# Example in IL

```
.class private auto ansi sealed Example
       extends [mscorlib]System.Object
{
  .custom instance void [Microsoft.VisualBasic]Microsoft.VisualBasic.
      CompilerServices.StandardModuleAttribute::.ctor() = ( 01 00 00 00 )
  .field private static int32 theResult
  .method public static int32  Fibonacci(int32 x) cil managed
  {
    // Code size       31 (0x1f)
    .maxstack  3
    .locals init (int32 V_0)
    IL_0000:  ldarg.0
    IL_0001:  ldc.i4.0
    IL_0002:  ceq
    IL_0004:  ldarg.0
    IL_0005:  ldc.i4.1
    IL_0006:  ceq
    IL_0008:  or
    IL_0009:  brfalse.s  IL_000d

    IL_000b:  ldarg.0
    IL_000c:  ret

    IL_000d:  ldarg.0
    IL_000e:  ldc.i4.1
    IL_000f:  sub.ovf
    IL_0010:  call       int32 Example::Fibonacci(int32)
    IL_0015:  ldarg.0
    IL_0016:  ldc.i4.2
    IL_0017:  sub.ovf
    IL_0018:  call       int32 Example::Fibonacci(int32)
    IL_001d:  add.ovf
    IL_001e:  ret
  } // end of method Example::Fibonacci

  .method public static void  Main() cil managed
  {
    .entrypoint
    .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() = ( 01 00 00
        00 )
    // Code size       12 (0xc)
    .maxstack  8
    IL_0000:  ldc.i4.4
    IL_0001:  call       int32 Example::Fibonacci(int32)
    IL_0006:  stsfld     int32 Example::theResult
    IL_000b:  ret
  } // end of method Example::Main

} // end of class Example
```

Listing A.4: Fibonacci VB.NET Example in IL

Appendix B

Calculator Example: IL Code and ASG

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 2.0.50727.42
//  Copyright (c) Microsoft Corporation.  All rights reserved.

// Metadata version: v2.0.50727
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z\V.4..
  .ver 2:0:0:0
}
.assembly calc_inheritance
{
  .custom instance void [mscorlib]System.Runtime.CompilerServices.
      CompilationRelaxationsAttribute::.ctor(int32) = ( 01 00 08 00 00 00 00 00 )
  .custom instance void [mscorlib]System.Runtime.CompilerServices.
      RuntimeCompatibilityAttribute::.ctor() = ( 01 00 01 00 54 02 16 57 72 61 70 4
      E 6F 6E 45 78   // ....T..WrapNonEx                                    63
      65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )       // ceptionThrows.
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module calc_inheritance.exe
// MVID: {A3A6DEF5-33D0-4757-B3EB-4E3A503588D2}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x02E90000

// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit Calc
       extends [mscorlib]System.Object
{
  .method public hidebysig newslot virtual
          instance int32  apply(int32 a,
                                int32 b) cil managed
  {
    // Code size       4 (0x4)
    .maxstack  8
    IL_0000:  ldarg.1
    IL_0001:  ldarg.2
    IL_0002:  add
    IL_0003:  ret
  } // end of method Calc::apply

```

```
43    .method public hidebysig specialname rtspecialname
44            instance void  .ctor() cil managed
45    {
46      // Code size       7 (0x7)
47      .maxstack  8
48      IL_0000:  ldarg.0
49      IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
50      IL_0006:  ret
51    } // end of method Calc::.ctor
52
53  } // end of class Calc
54
55  .class private auto ansi beforefieldinit CalcAdd
56          extends Calc
57  {
58    .method public hidebysig specialname rtspecialname
59            instance void  .ctor() cil managed
60    {
61      // Code size       7 (0x7)
62      .maxstack  8
63      IL_0000:  ldarg.0
64      IL_0001:  call       instance void Calc::.ctor()
65      IL_0006:  ret
66    } // end of method CalcAdd::.ctor
67
68  } // end of class CalcAdd
69
70  .class private auto ansi beforefieldinit CalcDiv
71          extends Calc
72  {
73    .method public hidebysig virtual instance int32
74            apply(int32 a,
75                  int32 b) cil managed
76    {
77      // Code size       4 (0x4)
78      .maxstack  8
79      IL_0000:  ldarg.1
80      IL_0001:  ldarg.2
81      IL_0002:  div
82      IL_0003:  ret
83    } // end of method CalcDiv::apply
84
85    .method public hidebysig specialname rtspecialname
86            instance void  .ctor() cil managed
87    {
88      // Code size       7 (0x7)
89      .maxstack  8
90      IL_0000:  ldarg.0
91      IL_0001:  call       instance void Calc::.ctor()
92      IL_0006:  ret
93    } // end of method CalcDiv::.ctor
94
95  } // end of class CalcDiv
96
97  .class private auto ansi beforefieldinit CalcMean
98          extends Calc
99  {
100   .method public hidebysig virtual instance int32
101           apply(int32 a,
102                 int32 b) cil managed
103   {
104     // Code size       22 (0x16)
105     .maxstack  4
106     .locals init (class Calc V_0)
107     IL_0000:  newobj     instance void CalcDiv::.ctor()
108     IL_0005:  stloc.0
109     IL_0006:  ldloc.0
```

```
110     IL_0007:  ldarg.0
111     IL_0008:  ldarg.1
112     IL_0009:  ldarg.2
113     IL_000a:  call       instance int32 Calc::apply(int32,
114                                                     int32)
115     IL_000f:  ldc.i4.2
116     IL_0010:  callvirt   instance int32 Calc::apply(int32,
117                                                     int32)
118     IL_0015:  ret
119   } // end of method CalcMean::apply
120
121   .method public hidebysig specialname rtspecialname
122           instance void  .ctor() cil managed
123   {
124     // Code size       7 (0x7)
125     .maxstack  8
126     IL_0000:  ldarg.0
127     IL_0001:  call       instance void Calc::.ctor()
128     IL_0006:  ret
129   } // end of method CalcMean::.ctor
130
131 } // end of class CalcMean
132
133 .class private auto ansi beforefieldinit Test
134        extends [mscorlib]System.Object
135 {
136   .field private static int32 x
137   .field private static int32 y
138   .field private static int32 z
139   .method public hidebysig static void  Main() cil managed
140   {
141     .entrypoint
142     // Code size       63 (0x3f)
143     .maxstack  3
144     .locals init (int32 V_0,
145              int32 V_1,
146              class Calc V_2)
147     IL_0000:  ldc.i4.s   10
148     IL_0002:  stloc.0
149     IL_0003:  ldc.i4.2
150     IL_0004:  stloc.1
151     IL_0005:  newobj     instance void CalcAdd::.ctor()
152     IL_000a:  stloc.2
153     IL_000b:  ldloc.2
154     IL_000c:  ldloc.0
155     IL_000d:  ldloc.1
156     IL_000e:  callvirt   instance int32 Calc::apply(int32,
157                                                     int32)
158     IL_0013:  stsfld     int32 CalcMain::x
159     IL_0018:  newobj     instance void CalcDiv::.ctor()
160     IL_001d:  stloc.2
161     IL_001e:  ldloc.2
162     IL_001f:  ldloc.0
163     IL_0020:  ldloc.1
164     IL_0021:  callvirt   instance int32 Calc::apply(int32,
165                                                     int32)
166     IL_0026:  stsfld     int32 CalcMain::y
167     IL_002b:  newobj     instance void CalcMean::.ctor()
168     IL_0030:  stloc.2
169     IL_0031:  ldloc.2
170     IL_0032:  ldloc.0
171     IL_0033:  ldloc.1
172     IL_0034:  callvirt   instance int32 Calc::apply(int32,
173                                                     int32)
174     IL_0039:  stsfld     int32 CalcMain::z
175     IL_003e:  ret
176   } // end of method CalcMain::Main
```

```
177
178   .method public hidebysig specialname rtspecialname
179           instance void  .ctor() cil managed
180   {
181     // Code size       7 (0x7)
182     .maxstack  8
183     IL_0000:  ldarg.0
184     IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
185     IL_0006:  ret
186   } // end of method CalcMain::.ctor
187
188 } // end of class CalcMain
```

Listing B.1: IL code for Calculator Example

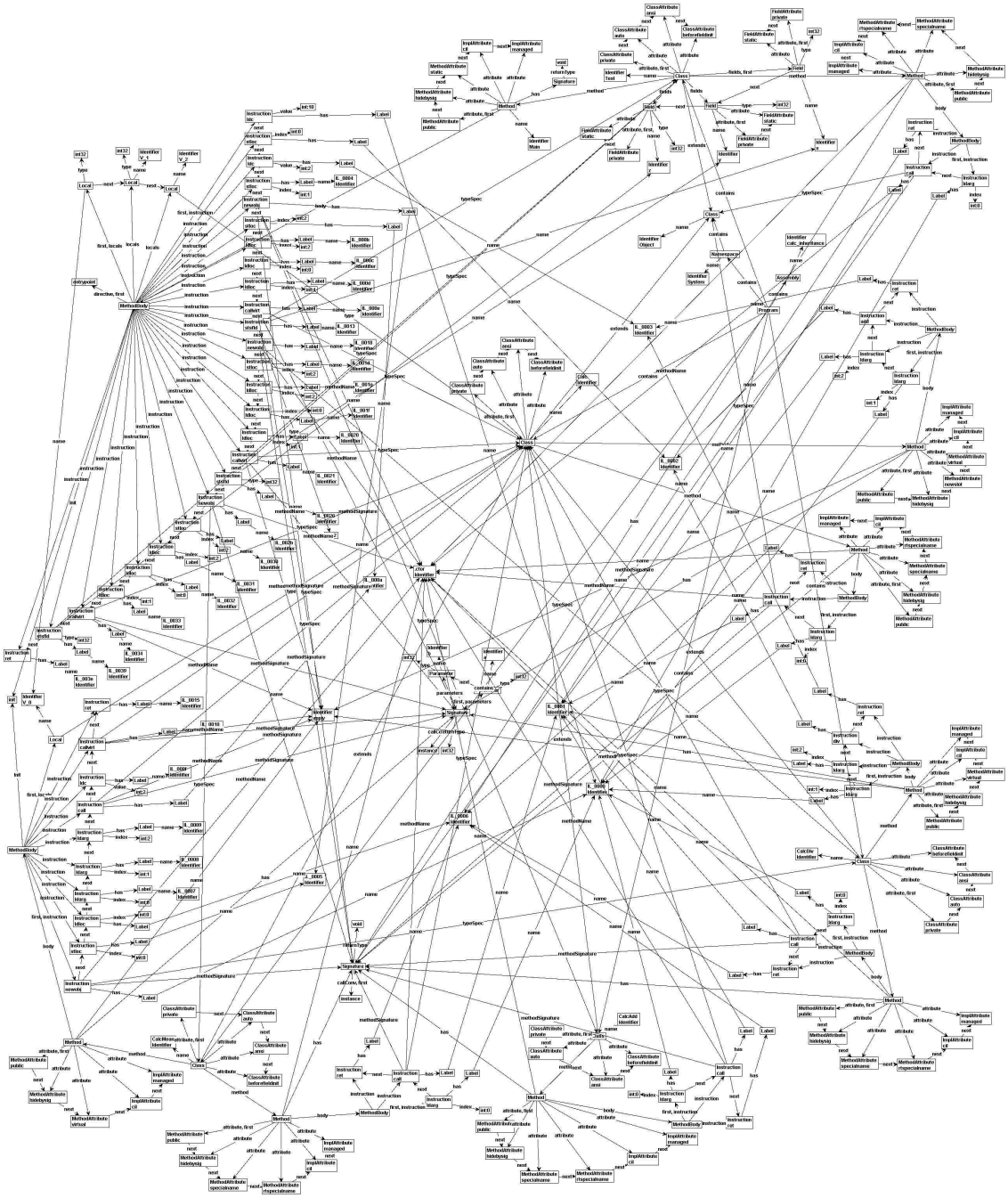


Figure B.1: ASG Calculator Example

Appendix C

Production Rules - Simulation

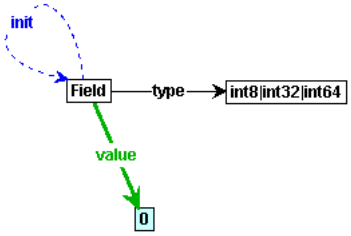


Figure C.1: 1.init_field_int

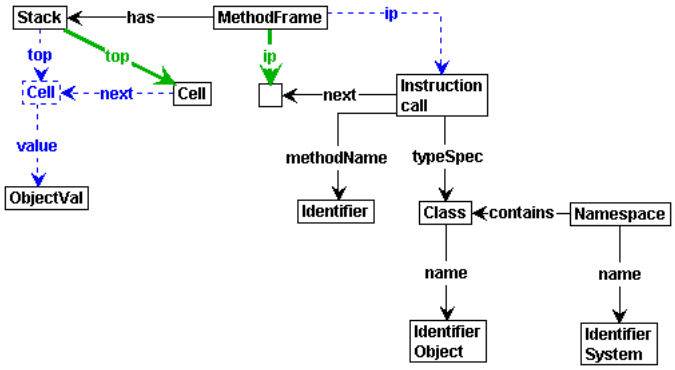


Figure C.2: 1.instr_call_system_object

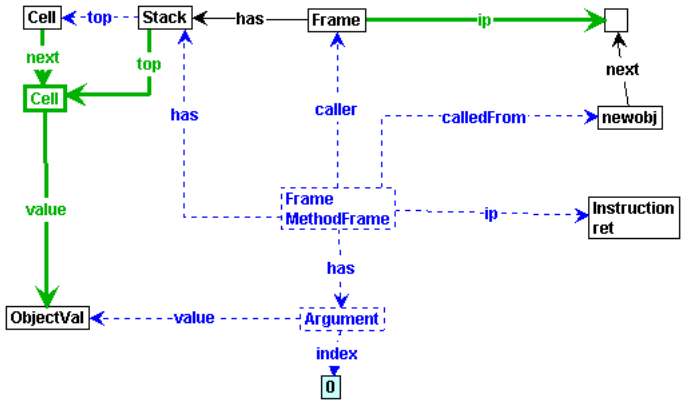


Figure C.3: 1.instr_ret_newobj

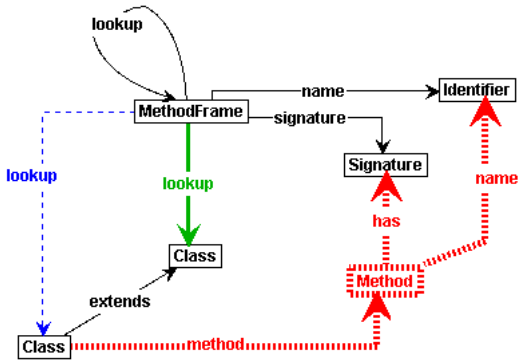


Figure C.4: callvirt_propagate

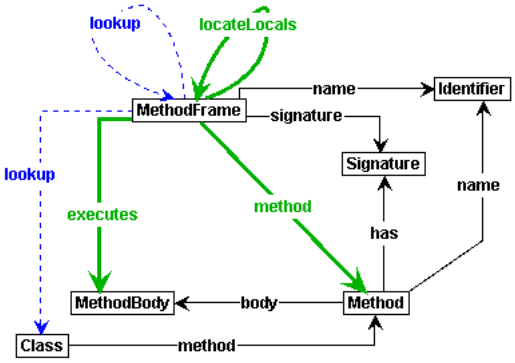


Figure C.5: callvirt_resolve

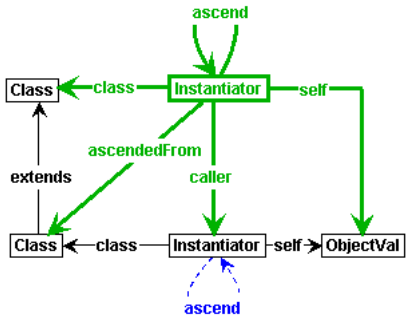


Figure C.6: class_ascend

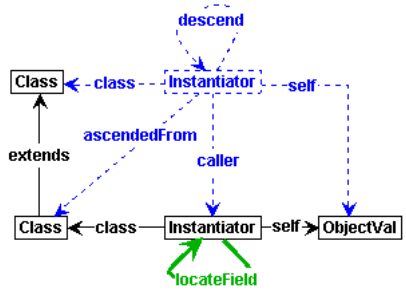


Figure C.7: class_descend

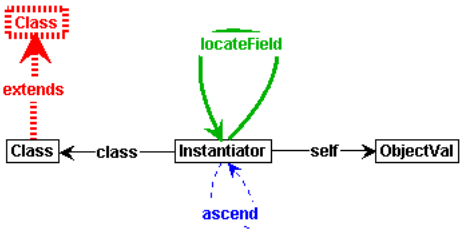


Figure C.8: class_initialize

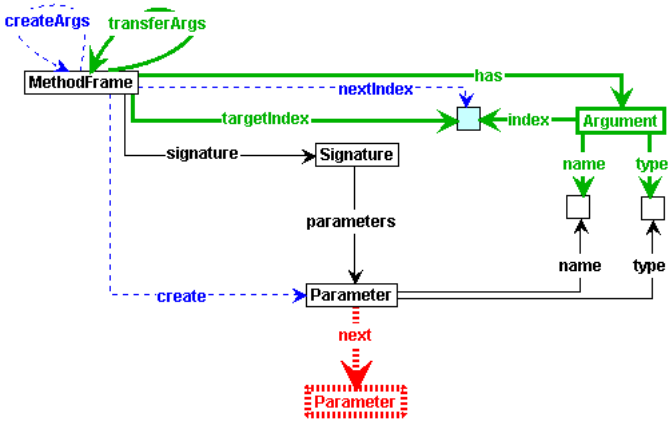


Figure C.9: create_args_last

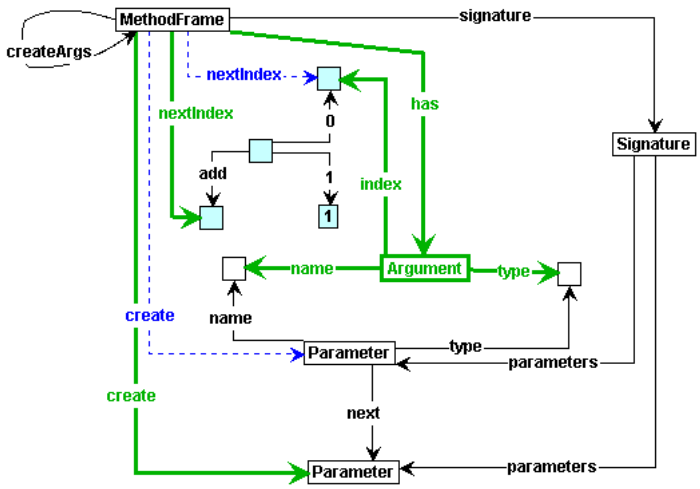


Figure C.10: create_args_next

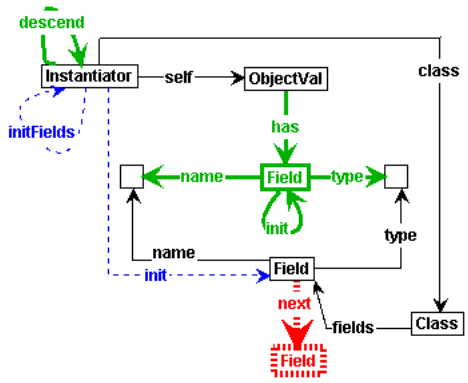


Figure C.11: init_fields_last

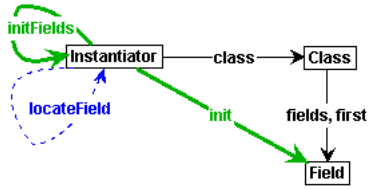


Figure C.12: init_fields_locate_first

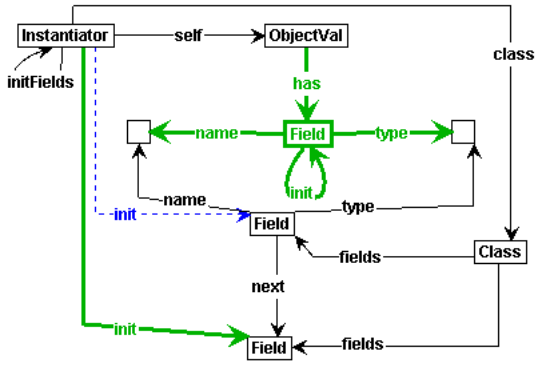


Figure C.13: init_fields_next

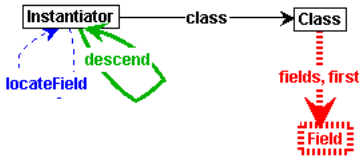


Figure C.14: init_fields_none

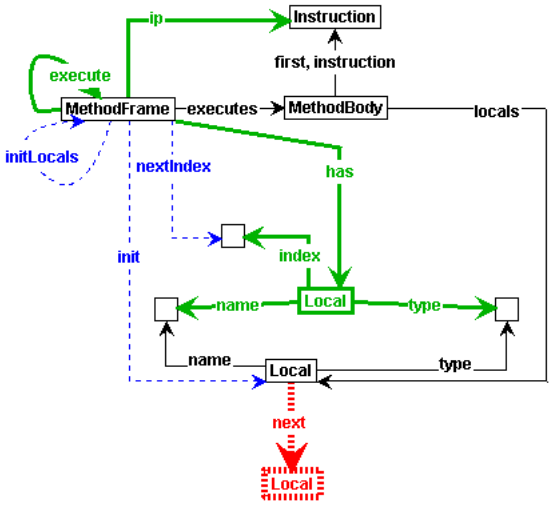


Figure C.15: init_locals_last

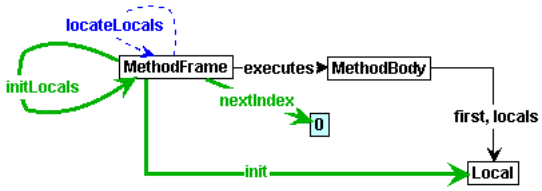


Figure C.16: init_locals_locate_first

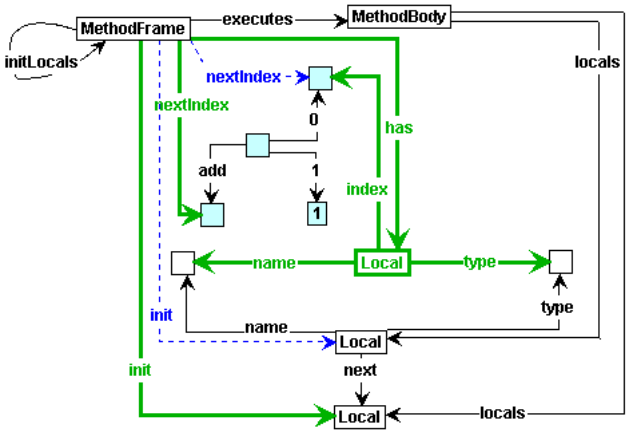


Figure C.17: init_locals_next

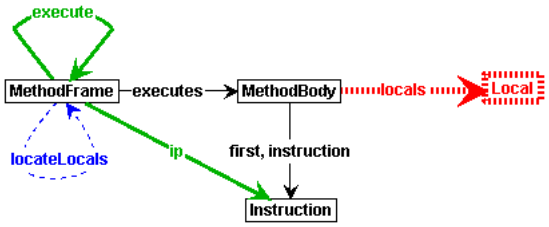


Figure C.18: init_locals_locate_none

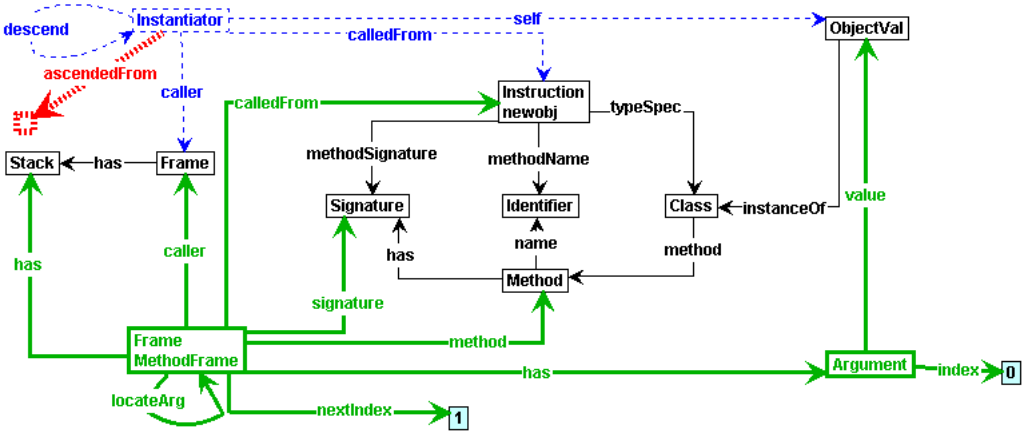


Figure C.19: instantiator_constr

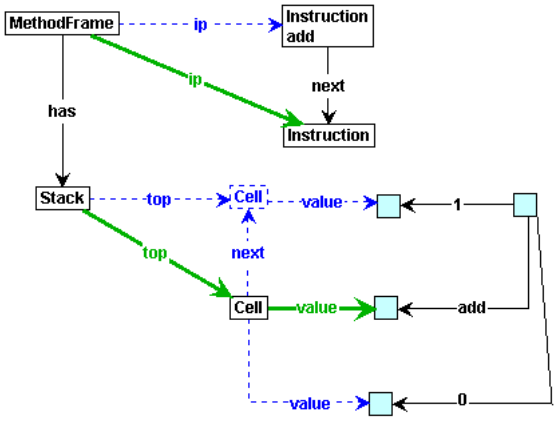


Figure C.20: instr_add

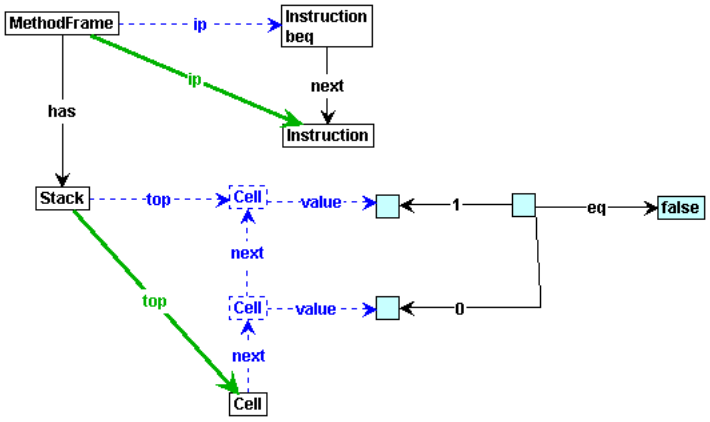


Figure C.21: instr_beq_false

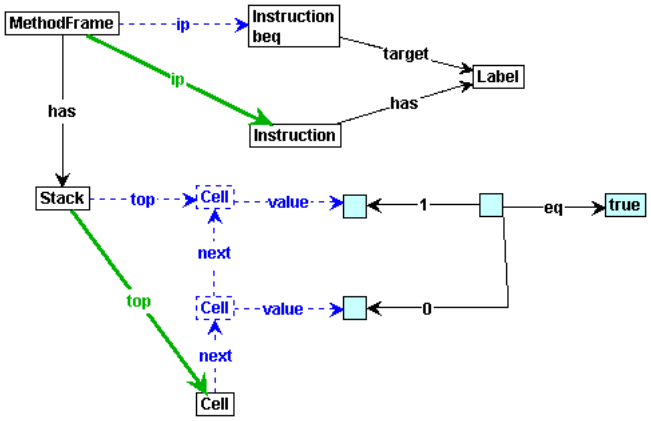


Figure C.22: instr_beq_true

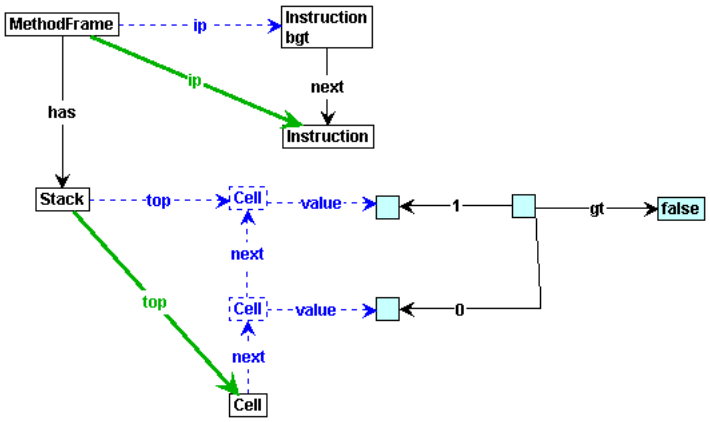


Figure C.23: instr_bgt_false

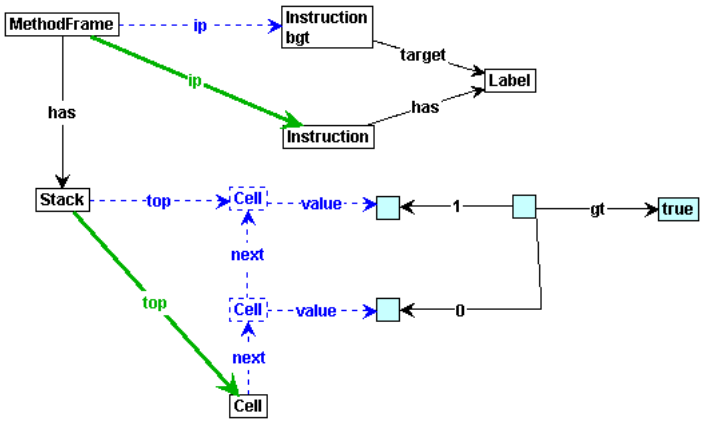


Figure C.24: instr_bgt_true

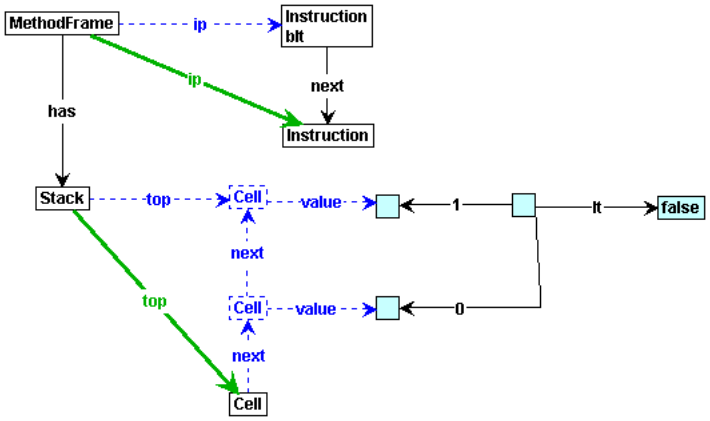


Figure C.25: instr_blt_false

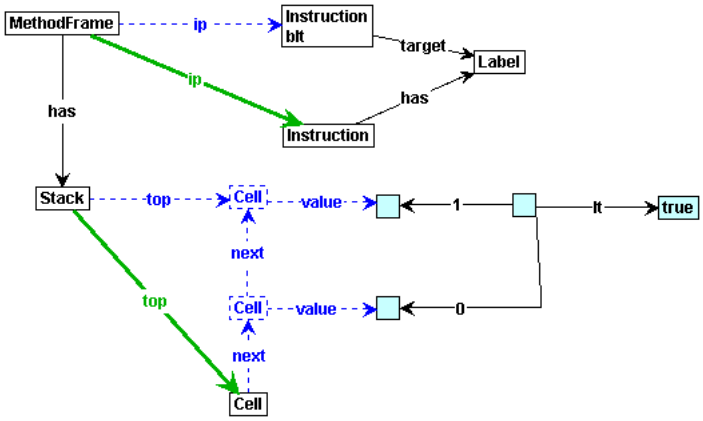


Figure C.26: instr_blt_true

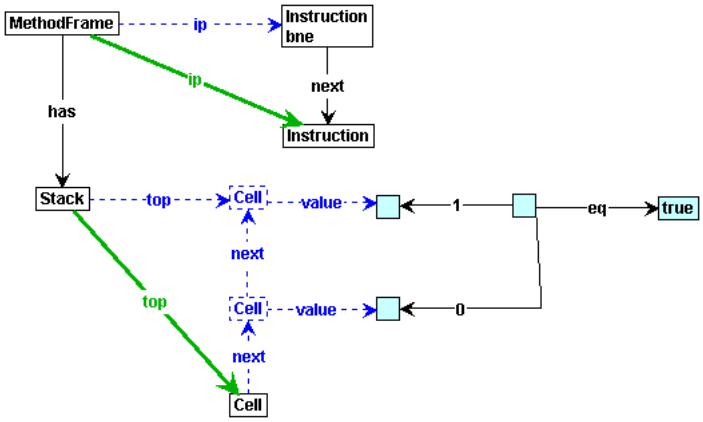


Figure C.27: instr_bne_false

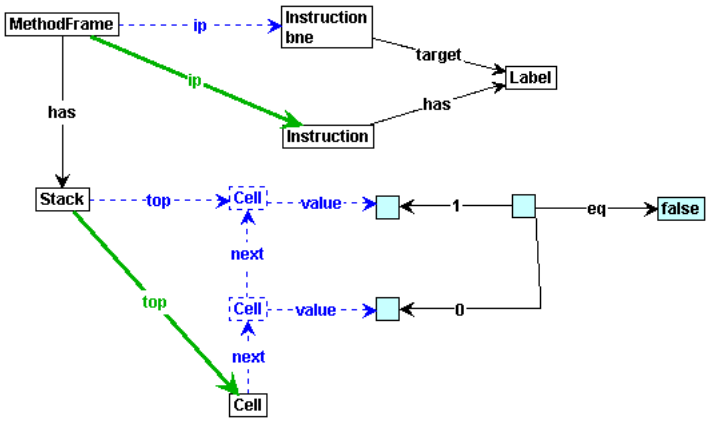


Figure C.28: instr_bne_true

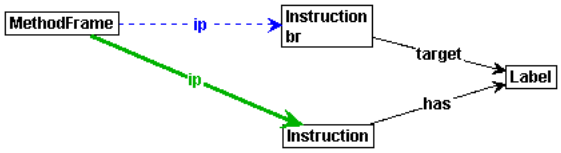


Figure C.29: instr_br

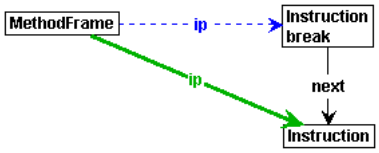


Figure C.30: instr_break

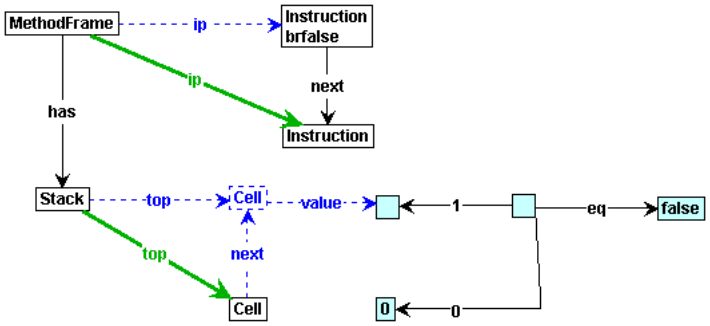


Figure C.31: instr_brfalse_false

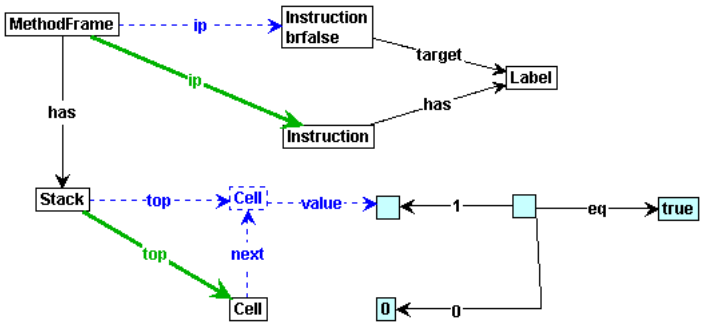


Figure C.32: instr_brfalse_true

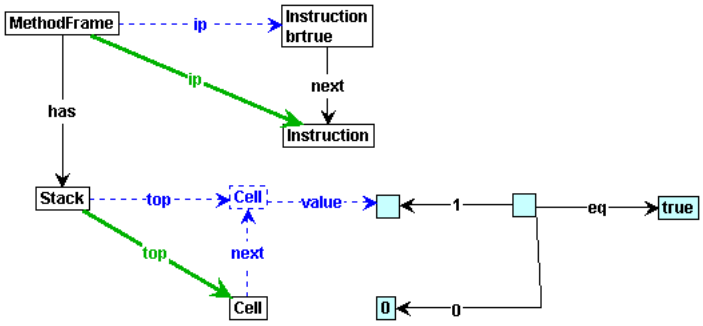


Figure C.33: instr_brtrue_false

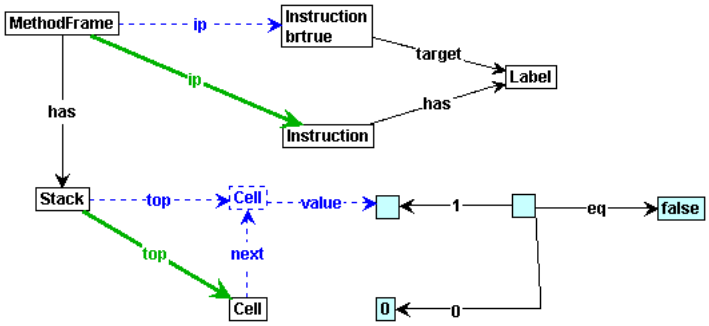


Figure C.34: instr_brtrue_true

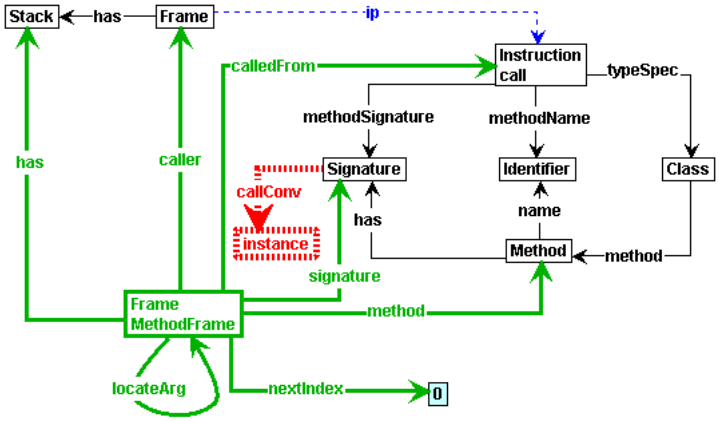


Figure C.35: instr_call

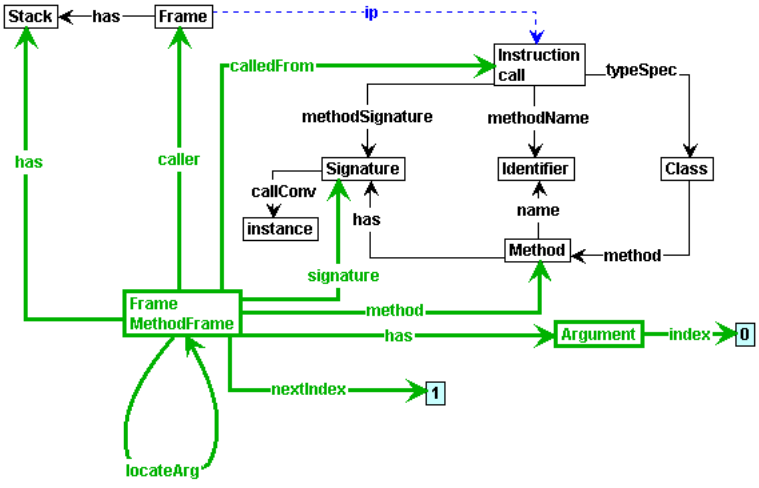


Figure C.36: instr_call_instance

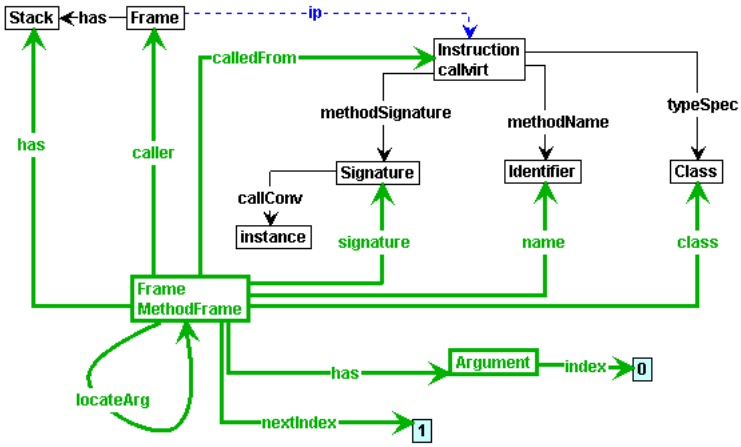


Figure C.37: instr_callvirt

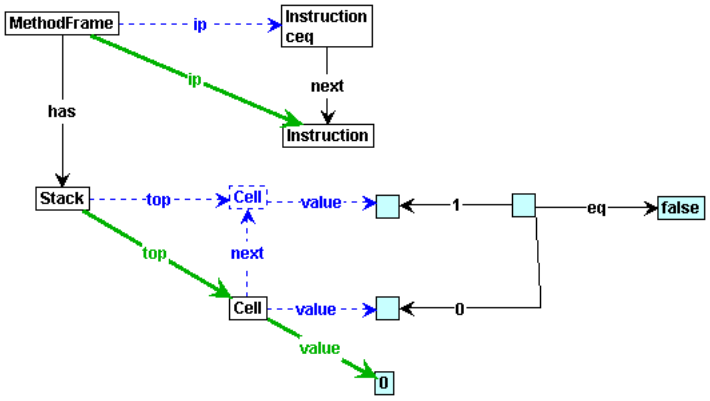


Figure C.38: instr_ceq_false

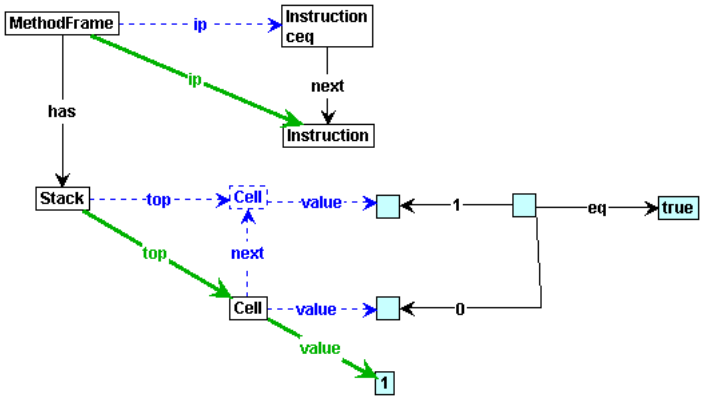


Figure C.39: instr_ceq_true

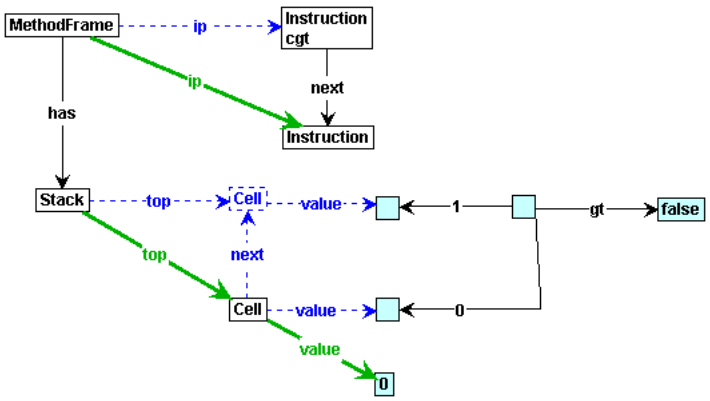


Figure C.40: instr_cgt_false

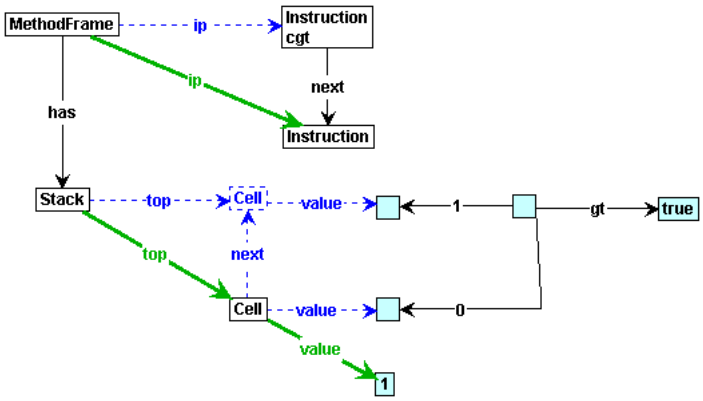


Figure C.41: instr_cgt_true

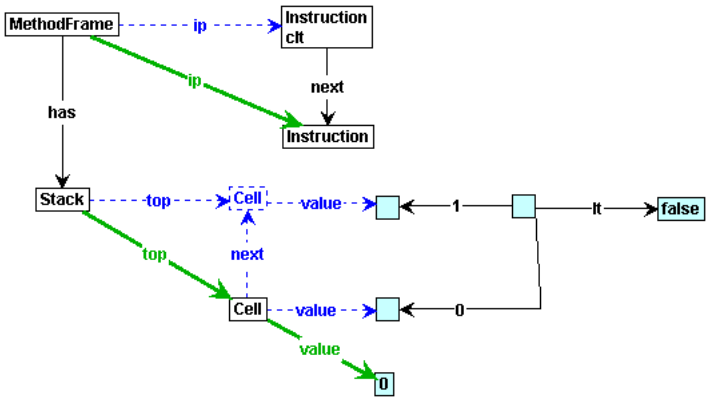


Figure C.42: instr_clt_false

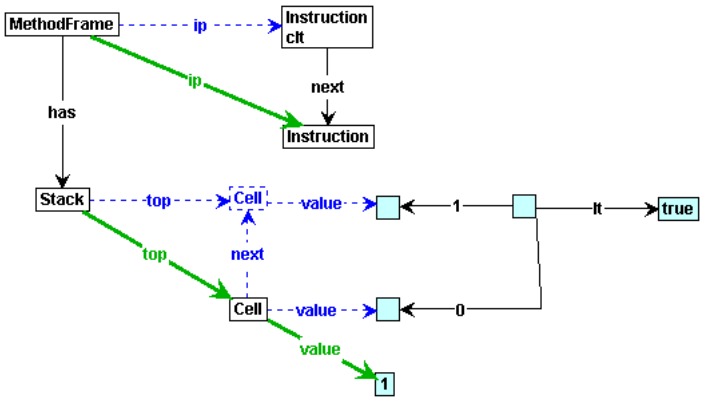


Figure C.43: instr_clt_true

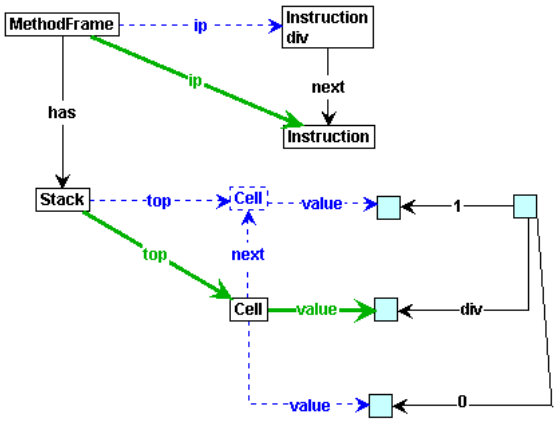


Figure C.44: instr_div

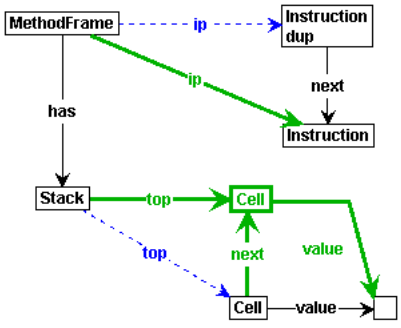


Figure C.45: instr_dup

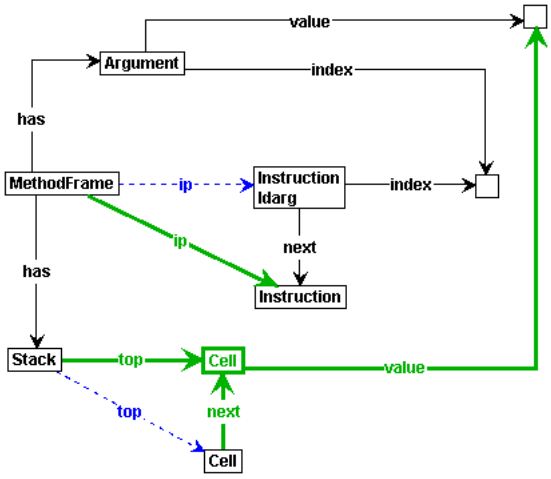


Figure C.46: instr_ldarg

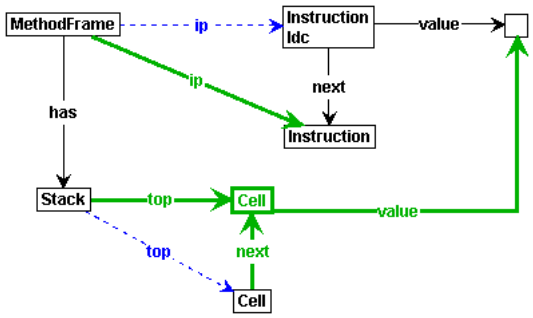


Figure C.47: instr_ldc

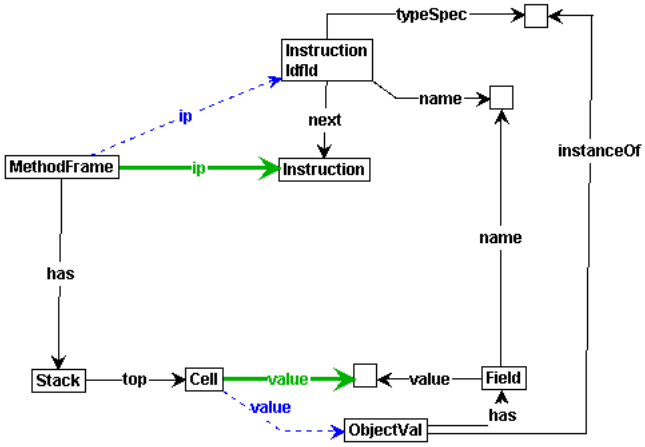


Figure C.48: instr_ldfld

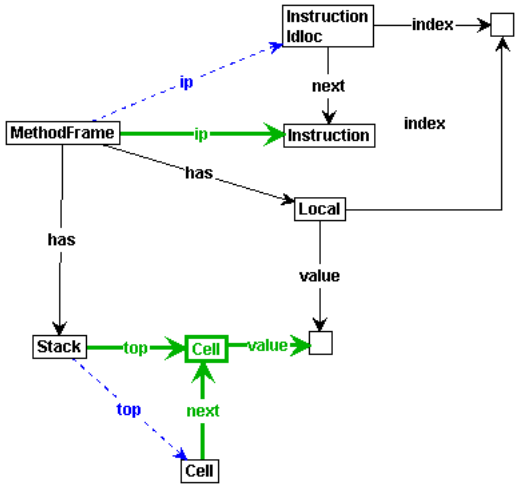


Figure C.49: instr_ldloc

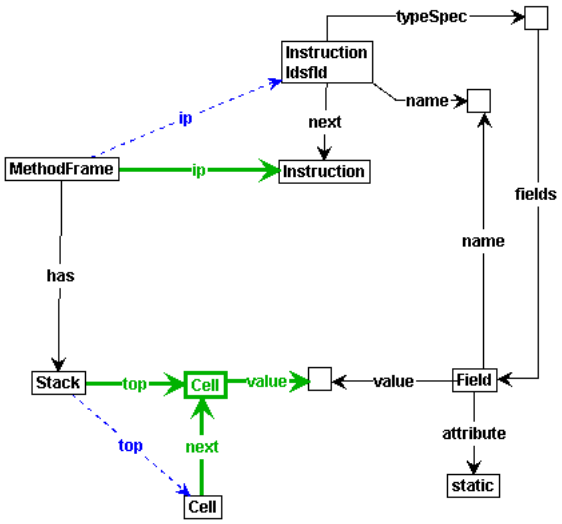


Figure C.50: instr_ldsfld

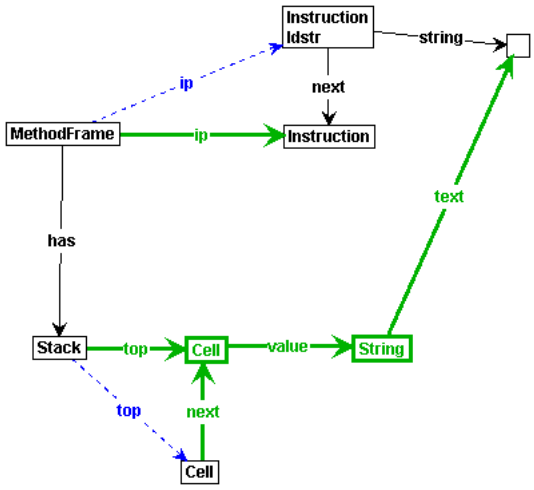


Figure C.51: instr_ldstr

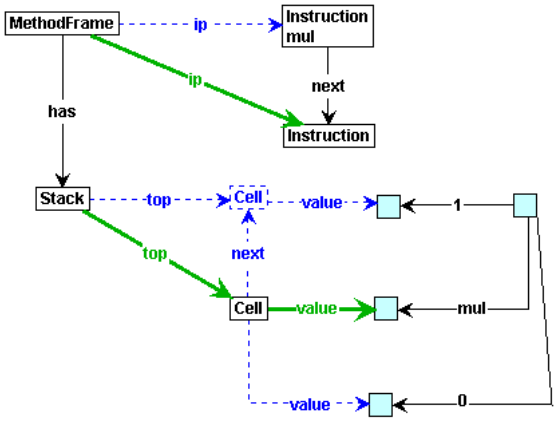


Figure C.52: instr_mul

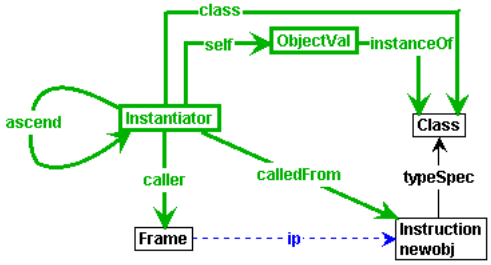


Figure C.53: instr_newobj

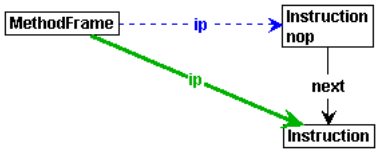


Figure C.54: instr_nop

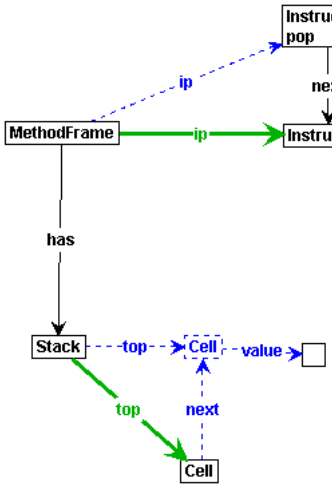


Figure C.55: instr_pop

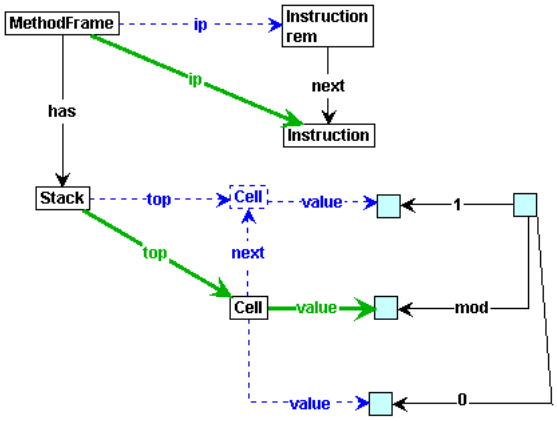


Figure C.56: instr_rem

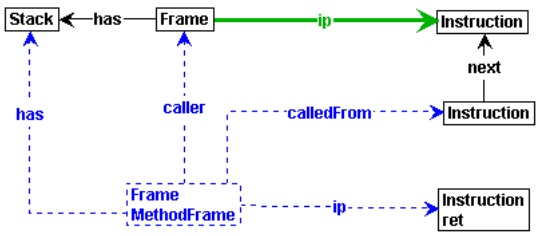


Figure C.57: instr_ret

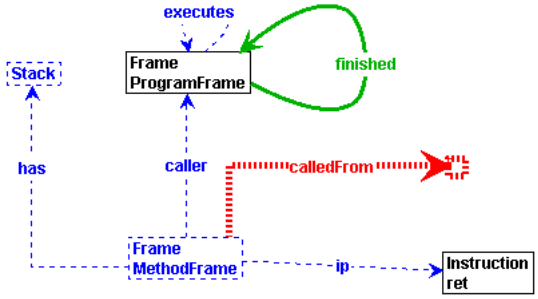


Figure C.58: instr_ret_program

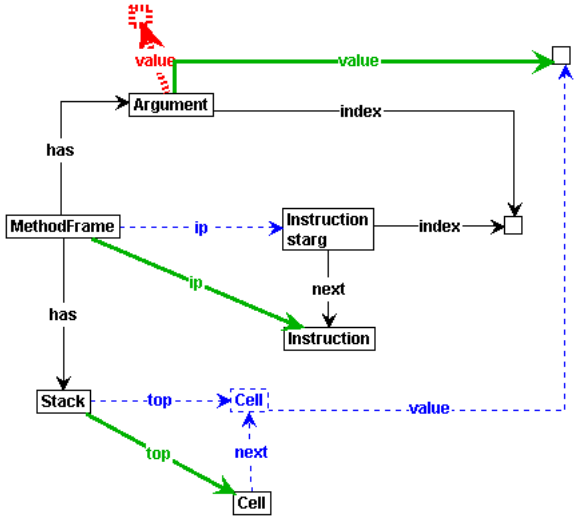


Figure C.59: instr_starg

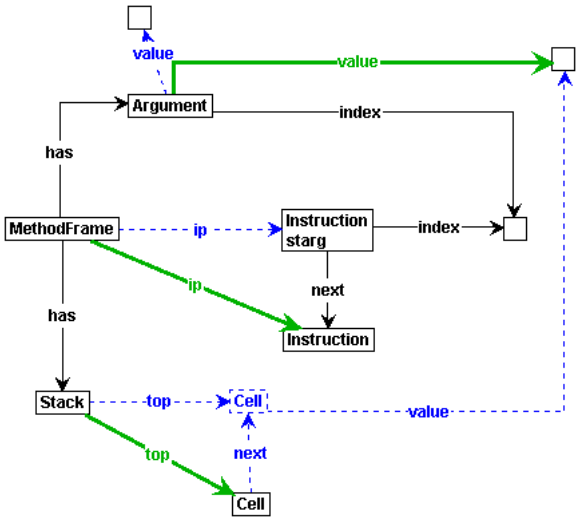


Figure C.60: instr_starg_new_value

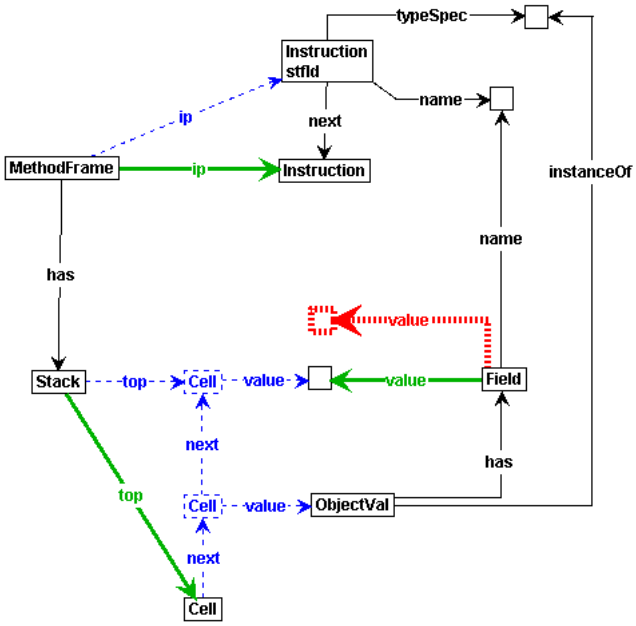


Figure C.61: instr_stfld

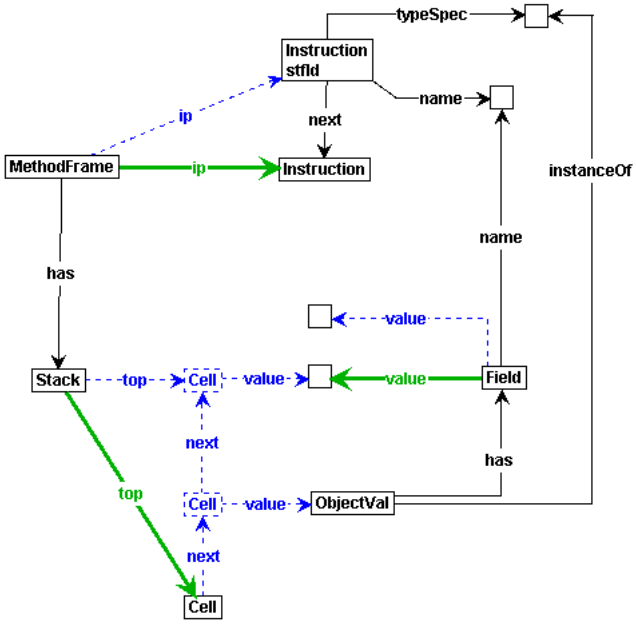


Figure C.62: instr_stfld_new_value

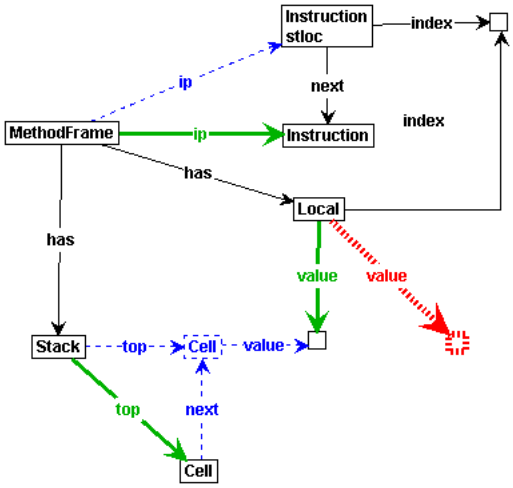


Figure C.63: instr_stloc

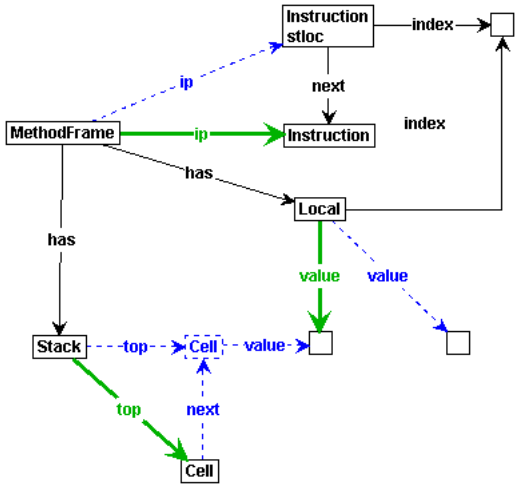


Figure C.64: instr_stloc_new_value

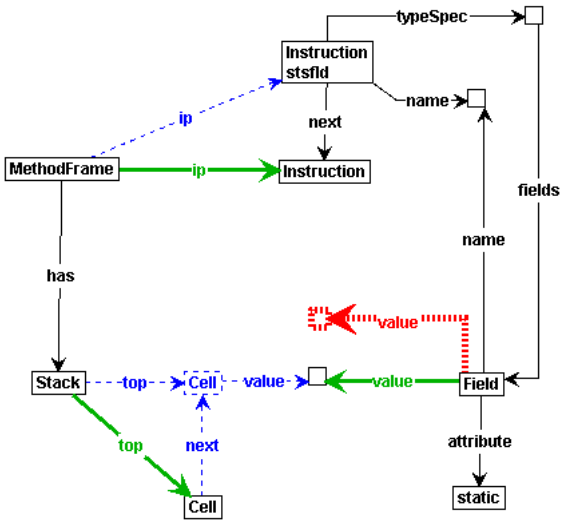


Figure C.65: instr_stsfld

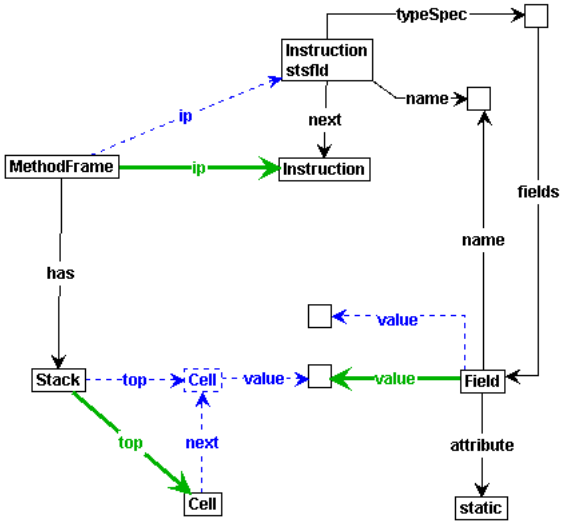


Figure C.66: instr_stsfld_new_value

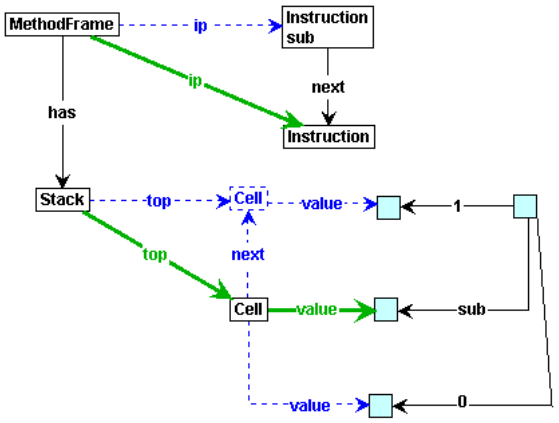


Figure C.67: instr_sub

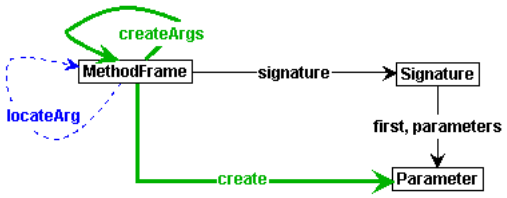


Figure C.68: locate_args_first

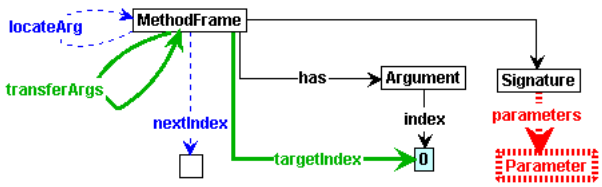


Figure C.69: locate_args_instance

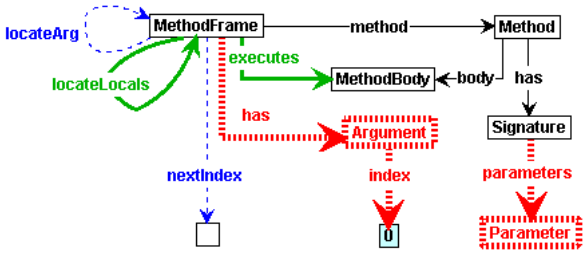


Figure C.70: locate_args_none

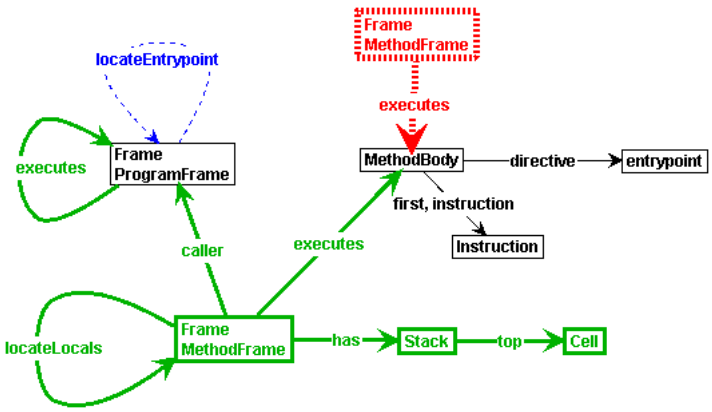


Figure C.71: methodframe_entrypoint

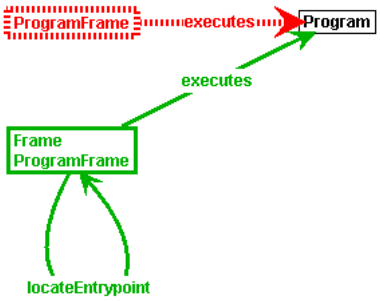


Figure C.72: program

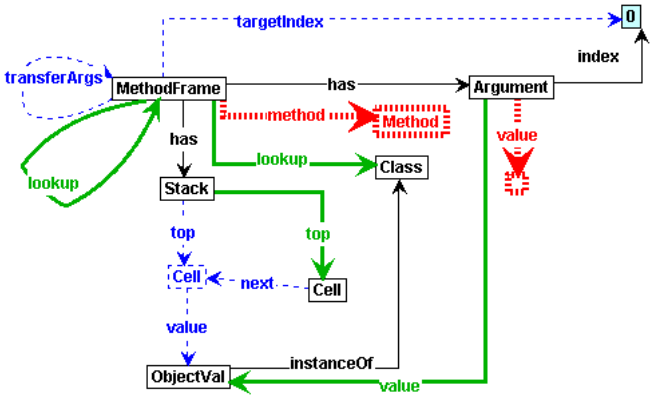


Figure C.73: transfer_args_last_dynamic

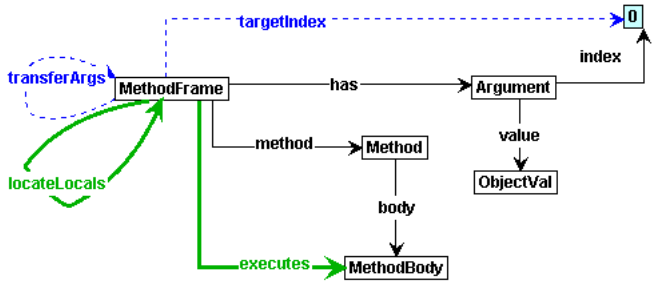


Figure C.74: transfer_args_last_newobj

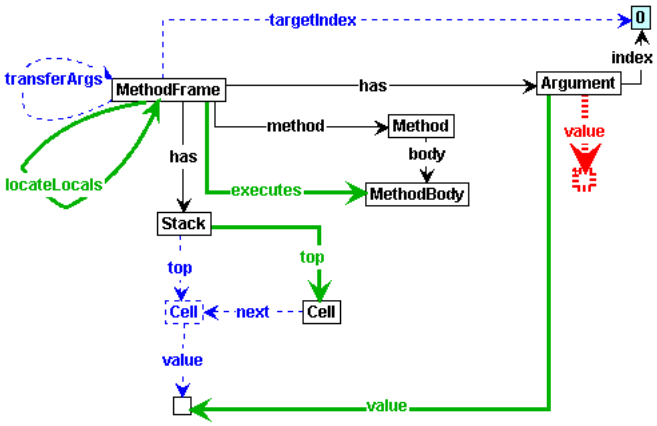


Figure C.75: transfer_args_last_static

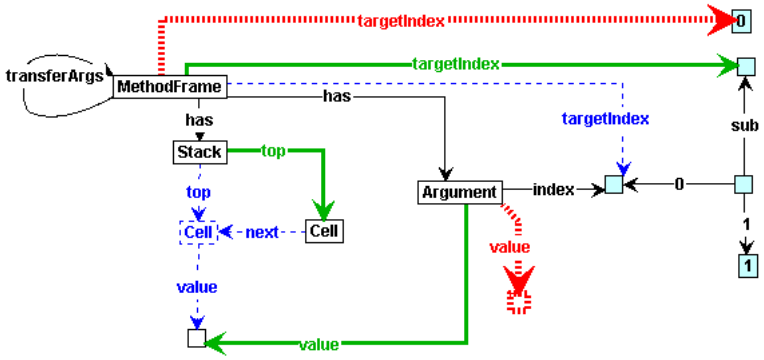


Figure C.76: transfer_args_previous