eXtensible Multi Security: Infrastruktura bezpieczeństwa dla platformy .NET

Wiktor Zychla
Praca doktorska

Promotor: Prof. Leszek Pacholski

Instytut Informatyki
Uniwersytet Wrocławski
ul. Joliot-Curie 15
50-383 Wrocław

Wrocław 2008

eXtensible Multi Security: Security Framework for .NET

Wiktor Zychla
PhD Thesis

Supervisor: Prof. Leszek Pacholski

Institute of Computer Science
University of Wroclaw
ul. Joliot-Curie 15
50-383 Wroclaw

Wroclaw 2008

Abstract

Modern computer systems depend on many pieces of software gathered together to perform certain activities. That is why various forms of distributed systems are being developed where individual software components come from different developers.

Such distributed systems bring a lot of freedom and convenience but when misused they can do a lot of damage. Thus, there is a deep need for various kinds of *safety* and *security* at different levels of a software life-cycle.

In this work we investigate some notions of safety taking the safety based on contracts into special consideration. We build eXtensible Multi Security - a framework based on the notion of Proof Carrying Code which is a powerful and coherent platform able to unify various notions of safety. We also show how XMS forms a certification framework for Microsoft Intermediate Language and other programming languages of the .NET Platform.

Declaration

This dissertation is the result of my own research.

Acknowledgement

This work was partially supported by Polish Ministry of Science and Higher Education grant 3 T11C 042 30

Contents

1 Introduction **1**
- 1.1 Convenience of distributed systems . . . 1
- 1.2 Safety and security of distributed systems . . . 1
- 1.3 eXtensible Multi Security . . . 2
- 1.4 Benefits of XMS . . . 2
- 1.5 Other Security Frameworks . . . 3
- 1.6 Overview of the dissertation . . . 5

2 Safety and security **7**
- 2.1 Enforcing safety policies . . . 7
- 2.2 Classification of safety properties . . . 7
- 2.3 Static and dynamic security . . . 8
- 2.4 Modularization and compositionality . . . 9

3 Core Paradigms **11**
- 3.1 Design By Contract . . . 11
  - 3.1.1 Overview of the Paradigm . . . 11
  - 3.1.2 Contracts in Practice . . . 12
- 3.2 Proof-Carrying Code . . . 12
  - 3.2.1 Overview . . . 12

4 The Intermediate Language **17**
- 4.1 The Runtime Environment . . . 18
  - 4.1.1 Managed Modules . . . 18
  - 4.1.2 Execution Process . . . 19
- 4.2 Safety and Security of IL . . . 19
  - 4.2.1 Safety . . . 20
  - 4.2.2 Security . . . 22
- 4.3 The Language . . . 23
  - 4.3.1 Naming Conventions . . . 24
  - 4.3.2 Types (classes) . . . 24
  - 4.3.3 Inheritance . . . 25
  - 4.3.4 Method bodies . . . 25
- 4.4 The Semantics . . . 30
  - 4.4.1 Values . . . 30
  - 4.4.2 Memory . . . 31
  - 4.4.3 Instructions . . . 33

5 The Infrastructure **39**
5.1 The Safety Policy . . . 39
5.1.1 Specification language . . . 42
5.2 Dynamic Verification Engine . . . 44
5.2.1 Test-Driven Development . . . 44
5.2.2 Instrumentation of .NET Code . . . 45
5.2.3 Using the Engine . . . 52
5.3 Static Verification Engine . . . 53
5.3.1 Symbolic Evaluation . . . 53
5.3.2 Symbolic Evaluation Cases . . . 55
5.3.3 The Safety Theorem . . . 61

6 Towards High-Level Languages **67**
6.1 From MSIL to High-Level Languages . . . 67
6.2 Common Certificate Specification . . . 69
6.3 High-Level Compiler Translation Schemes . . . 70
6.3.1 Variable Ordering . . . 70
6.3.2 Assignments, Expressions . . . 73
6.3.3 Loops . . . 73
6.4 Other High-Level language features . . . 74
6.4.1 Class Invariants . . . 74
6.4.2 Properties, Indexers . . . 76
6.4.3 Delegates . . . 77

7 Practical Issues **81**
7.1 The Implementation . . . 81
7.1.1 Code-Producer Components . . . 81
7.1.2 Certification Components . . . 83
7.2 Private Computation . . . 83

8 Conclusion and Future Work **85**
8.1 Contribution . . . 85
8.2 Future Work . . . 85

A MSIL Instruction Set **97**

B The Soundness Theorem **103**

C Examples **107**
C.1 Dynamic Verification Engine . . . 107
C.2 Static Verification Engine . . . 111

Chapter 1

Introduction

1.1 Convenience of distributed systems

Distributed systems play a major role in today's computer systems.

Although we usually think of distributed systems as of huge commercial frameworks like banking systems, we use such systems every day often being unaware of it. Modern distributed systems integrate well even with common web browsers and are able to perform both *server-side* and *client-side* activity.

For example, when I visit my favourite online bookshop and I order some books, I perform some activity locally: I enter my personal data and I choose the books I like to buy. Then I click a button and there is some activity performed remotely: my data and my order are sent to the remote server which stores the order in a database.

In fact however, even these applications which we buy or even download for free can be seen as distributed systems. Indeed, when I install a new piece of software in my operating system, it can itself perform some local or remote activity and interacts with other software components. It can for example gather some data from my computer and use an external library to send the data somewhere with or without my knowledge.

The convenience and freedom offered by the distributed systems is sometimes misused. The software and the hardware is a potential victim to a malicious virus, the data is a potential victim to a trojan horse or a spy-software. There is a lot of carefreeness when dealing with distributed systems. There are still critical bugs found even in vital parts of operating systems and commonly used applications. From the developer's perspective it is still too easy to introduce unintentional bugs into the software or even more - to trick the trusting user and make him run a malicious code on his/her system. From the user's perspective it is often impossible to examine every aspect of the software.

1.2 Safety and security of distributed systems

Of course, every action has a reaction. That is why there is a lot of work at the area of safety and security of distributed systems.

Alas, over forty years after the Internet has been born, the majority of users still have to believe that the software they buy or download is safe in a sense that it will not do any harm to their hardware and data.

Widely spread antivirus software can detect several thousands of computer viruses. That's good. Alas, it is able to detect only these viruses that are known. That's bad. If the new virus is released, my machine is as vulnerable like a little baby.

Runtime environments can dynamically supervise the code execution and for example disallow the execution of some activities. That's good. They cannot however make sure that the code runs correctly. That's bad. Even the advanced managed code is not a bit helpful when the banking software altered by hackers steals money from my bank account.

1.3 eXtensible Multi Security

The goal of eXtensible Multi Security (XMS, [46]) is to unify various notions in one coherent and extensible platform. The idea is built around two component paradigms which together form a powerful certification framework - these two paradigms are Design By Contract (DBC, [26]) and Proof Carrying Code (PCC, [33]) unified around the Microsoft .NET Runtime Environment.

The original PCC approach focuses on type-safety. Alas, the type-safety does not guarantee that other important features of safety are preserved. In fact, various aspects of safety are rather independent. The code can be type-safe but not correct or type-unsafe but perfectly safe from 'control flow' point of view.

This is where the XMS starts. The infrastructure can encode the Design By Contract specifications for software components written for the .NET Runtime Environment. The specification is then verified using the dynamic verification engine or the static verification engine. In case of the former the specification stored in the binary meta-data is used at the run time to check if it is satisfied. In case of the latter digital certificates take the form of formal logic proofs and are stored in the meta-data so that they do not play any role in the code execution but instead they are used in the verification process.

Although the core of XMS is built at the low level of Microsoft Intermediate Language (MSIL), the intermediate language of the .NET Platform, we show how the certificates can be adopted to high-level .NET languages like C# or VB.NET.

It is interesting to recall that although these two terms, safety and security are closely related there is a subtle difference. A software is *safe* when it will not do any harm to the user. A software is *secure* when some additional activity was taken to prevent the user from getting any harm from it. That is why we will say that on the one hand XMS helps us to build safe software and on the other hand it builds a security framework that is able to detect unsafe software.

1.4 Benefits of XMS

Here is the short summary of XMS benefits:

- XMS is designed to certify the MSIL language, one of the most widely used enterprise intermediate languages
- XMS certificates can be seamlessly adopted to high-level .NET languages
- XMS is primarily designed to certify the Design By Contract but other formal policies could also be expressed

- The certification framework makes it much easier for developers to find bugs in the software
- XMS certificates are built around the notion of PCC thus inheriting all desirable properties of PCC:
  - formal theorems guarantee that certificates cannot be compromised
  - the certificates are sufficient to guarantee that the code is valid, the authority of a code producer is completely insignificant
- XMS certificates are stored in meta-data so the proofs are bound with the code
- To support XMS the .NET Runtime Environment does not need to be changed in any way.

1.5 Other Security Frameworks

The PCC is not the only formal approach to notions of safety and security. There are dozens of interesting formalisms such as Model Checking and the μ-Calculus [24], Propositional Dynamic Logic (PDL, [8]), Model-Carrying Code [37], Symbolic Trajectory Evaluation (STE [17]), Communicating Sequential Processes [39] and Security Process Algebra [14], π-Calculus [29] and other Value-Passing Process Algebras ([19]) and many, many others which are beyond the scope of this work.

Both core components of XMS are subjects of extensive study. The Design By Contract evolved from earlier works of Hoare ([18]) and the research continues in many areas. In recent years the challenge of adopting the Hoare-style specifications into object-oriented languages has been taken by researchers and several approaches have been successful ([35], [45]). We believe that XMS is more general since it scales from the intermediate language to high-level languages and thus is not bound to a single programming language or a family of languages. Many other interesting works focus on selected aspects of the paradigm, for example on automatic discovering of loop invariants ([12]).

As for PCC, the original idea ([33]) focuses on type safety and is often compared to the Typed Assembly Language ([30]), a low-level infrastructure where type information is preserved during the compilation and is used to certify the type-safety of low-level code.

For many reasons the type safety is strongly desirable for untyped assembly-level languages. The language presented in the original PCC paper, SAL, is a good example - in [33] it is shown how the type-safety of SAL binaries can be enforced by using carefully designed logic that detect illegal memory I/O operations.

In such approach the primary goal of PCC is to validate the language compiler by detecting compile-time bugs. This idea was further adopted to certify the type safety of Java binaries at machine-level (SpecialJ compiler described in [7], [6]). Type safety policy has some major advantages - PCC signatures can be built automatically according to original signatures of methods and the proofs are rather straightforward. Thus the infrastructure can be automated to certify even large programs.

A more general approach to PCC was developed ([1] and [13]). In this approach the semantic properties of the language do not have to be known by the algorithm performing the analysis but are rather a part of the Safety Policy.

Other security properties can be enforced by PCC and some results that allow temporal specifications for a restricted low-level language were also shown ([5]).

From the distant perspective there are two aspects of PCC that must be addressed in each particular implementation:

- which language shall be certified?
- which Safety Policy shall be enforced?

Both these issues laid the base of XMS. XMS was designed to certify Microsoft Intermediate Language, widely recognized language targeted by a broad range of enterprise programming languages. But in contrast to for example the SpecialJ, XMS certifies the code at the MSIL level, not at the machine-level. We believe that the translation between MSIL and the actual machine-level language shall preserve all encoded safety properties.

Working at the level of intermediate language gives another benefit to the XMS - certificates can be used for any .NET high-level language (like C#). To make use of XMS a developer does not necessarily need to know MSIL but rather be aware of some compiler translation schemes.

On the yet another hand, the IL language **is** type-safe because that is the way it has been designed ([16]). What is more - the .NET Runtime Environment **type-checks** all binaries just before they are actually executed (binaries that pass the test are called *verifiable*). This static type-checking operation rejects binaries that do not follow strict type-safety rules. We will discuss this and other safety mechanisms of the .NET Runtime Environment in chapter 4.

Although the existing static algorithm for MSIL type-safety is potentially *weaker* than the PCC type-checking (for example, it does not aim at detecting invalid array indexing operations), the PCC style type-checking is beyond scope of XMS. Figure 1.1 shows precisely the level at which XMS operates and the level at which original PCC operates. While PCC focuses on type-safety of low level language, XMS focuses on safety of higher-level languages.

Initially XMS started as a PCC variant for a toy-like object language. After migration to .NET platform, XMS marks out its own way:

- XMS does not certify type-safety of the low level language but instead it allows to certify other safety policies of the MSIL language.
- Since the certificates can be applied to any high-level language, XMS is more general than solutions bound to a single low-level ([33]) or high-level ([36]) language.
- XMS will ultimately adopt other security policies, such as Non-Interference, to its verification engine

Currently, as a contract verification framework, XMS competes with specialized contract frameworks for .NET Platform like the Spec# ([28]). Below we list major differences between these two:

- Unlike XMS, Spec# is bound to a single language - it is a superset of C#.
- Unlike XMS, Spec# is bound to a single safety policy (contracts). XMS is an extensible framework with pluggable verification engines
- In Spec# contracts are declared using the language extensions and turned into inlined code during the compilation. In XMS, contracts are external to the language (attributes) and code instrumentation techniques are used for dynamic analysis

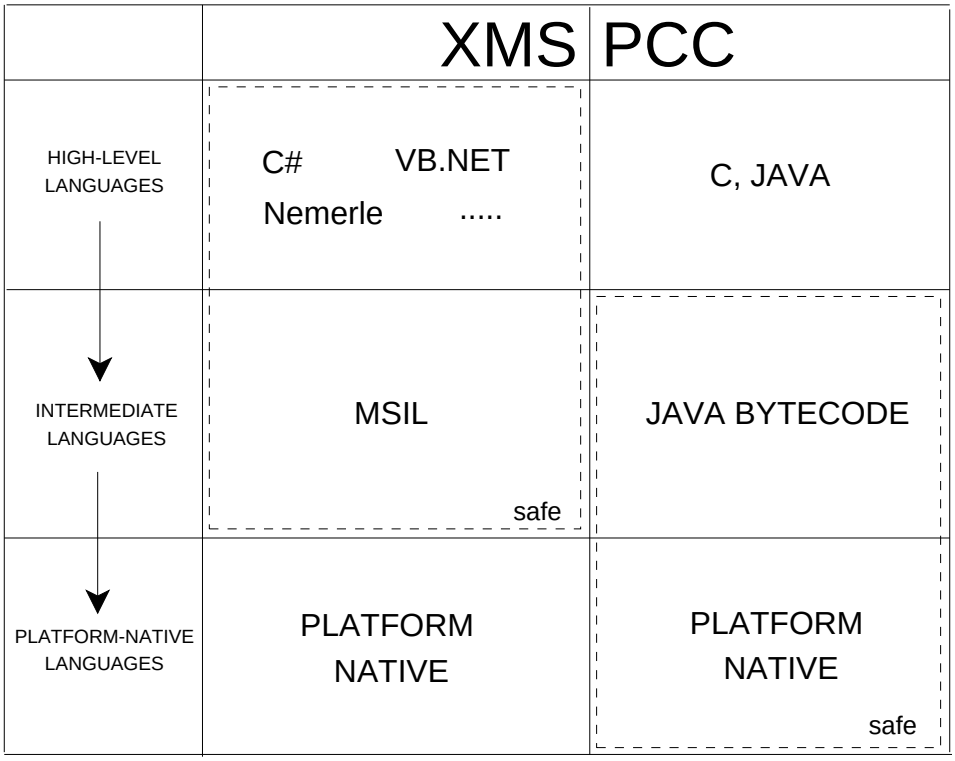

Figure 1.1: XMS safety versus PCC type-safety

- Spec# uses its own intermediate representation of the code, BoogiePL, which is interpreted and transformed before it is provided to the theorem prover. XMS uses symbolic evaluation to build verification traces directly from the .NET Intermediate Language code.

1.6 Overview of the dissertation

We start our dissertation with a brief introduction to the notion of **XMS** and motivate our work by comparing XMS to other security frameworks.

In the second chapter we present and classify various aspects of safety and security.

In the third chapter we describe two core XMS paradigms: the Design By Contract and Proof-Carrying Code. In the fourth chapter we present the core of XMS environment - the Microsoft Intermediate Language. Because the MSIL language was chosen as the target XMS language, we build the operational semantics of a significant subset of it. This subset captures the majority of important aspects of the IL including arithmetics, control flow, objects, arrays and exceptions.

Fifth chapter is a core of the dissertation. It is here where we present safety formalisms for the XMS, define a safety policy and show two XMS engines.

In the sixth chapter we show how MSIL level certificates can be seamlessly adopted to high-level .NET languages.

In the seventh chapter we discuss some internal details of XMS and discuss its practical applications.

Chapter 2

Safety and security

2.1 Enforcing safety policies

In this chapter we discuss several different notions of safety. It turns out that the short and seemingly precise term *safety* can have several different meanings. All of them are heavily studied and new solutions are constantly developed.

Talking about safety we will often use the term *safety policy*. The safety policy is a formal set of rules and restrictions that somehow tells us which programs are valid and which are invalid and should be considered illegal, unsafe.

An example of a naive safety policy would be

> *Any code that is given to me by system administrator is secure.*
> *Any other code is insecure.*

Of course such safety policy could bring a lot of harm and thus should not be probably considered seriously. Even in its much more complicated form known as **Authentication by Personal Authority**, such policy in fact does not state any security. These two terms, security and authentication, should not be mistaken. Even the strongest cryptographic signature does not guarantee that the code is secure, it can only authenticate the code producer. When one accepts a software component signed digitally by a well known and reliable software developer, does it mean that this new component is automatically safe? What if the software component is unintentionally insecure or worse, someone used the digital signature of this well known developer to sign an intentionally malicious software component?

The true security can be only forced by so called **Language Based Security**. It means that the safety policy must somehow exploit the semantics of the programming language, an operating system or the runtime environment.

The true security must then be objective; we must not *believe*, we have to *be sure*.

2.2 Classification of safety properties

What kind of reasonable security policies should be considered? Well, from a really distant perspective we can name following main categories of security properties:

memory safety where the code should not access the memory that was not assigned to it in an explicit way. This is one of primary security properties and because of its importance it is

built into all mature operating systems. The times where it was easy to write a malicious code that steals or modifies the memory of another process are, hopefully, gone forever.

type safety where the *types* play the main role. It appears that the strict typing discipline can eliminate many common errors like accessing resources that should be hidden or calling methods with wrong number of parameters or parameters of wrong type. These common errors are not only exploited by a malicious software but also are serious worries to code producers.

As a benefit, strict typing can make it easier to perform common code analysis, transformations and optimizations. An example of such system can be found in [30].

code correctness where the *secure* means *producing correct results*. Such notion of security should be desirable for example in financial or computational applications where a small mistake in a calculation could lead to unpredictable results.

An example of such approach is described in [2].

control flow safety where there should be no jumps outside the current block of code and any jump should target the beginning of valid code instruction. Also, it should not be possible to exploit the so called *buffer underrun* technique, where due to the corruption of runtime stack, the data supplied by the attacker is executed in the context of running application.

An example of such policy is implemented as the algorithm that statically verifies the integrity of .NET assemblies.

information flow safety aims at controlling the way the information flows among different entities. Such properties were primarily proposed to formally describe the notion of *confidentiality* but turned out to be usable in other situations as well for example to detect deadlocks or unwanted covert channels.

But how can these policies be verified? Which mechanisms can be used to distinguish between secure and insecure code? Well, in general there are two main approaches: static and dynamic. The distinction between these two is crucial because both have their own pros and cons.

2.3 Static and dynamic security

In *dynamic* checking, the safety policy is constantly checked to be valid at the run time. This of course requires the existence of a virtual machine or a runtime environment that would be powerful enough, in the sense that it can detect any activity that breaks the safety policy. An example of such an infrastructure is the Java Virtual Machine or the Microsoft .NET Framework. Both ”supervise” the code execution and enforce precise checks before any potentially dangerous instruction is executed.

For example, if the safety policy forbids any I/O activity, the runtime environment puts proper checks before each I/O call. If the safety policy forbids to make connections to some specific servers, again, proper checks are put into the code and the exception is thrown when an illegal connection is detected. What’s more, illegal memory operations are immediately detected.

The dynamic approach has undeniable advantages. It allows to define quite precise security policies and checks them very strictly. It also has drawbacks because it is not general. The existence of a runtime environment with rich standard library is a non-trivial condition and there is still a lot of code that runs natively on a target hardware. The dynamic approach suffers from something we could call a *boom* effect. Let's imagine that dynamically checked code runs for a long period of time and performs a lot of activity during all that time. Then, suddenly the safety policy becomes broken and the execution is terminated. But what will happen to the effects of all actions that were executed to that time? Should they be cancelled? Is it possible to do it at all? These questions do not have any satisfying answer.

On the other hand in the process of *static* checking the code is verified without being actually run. The answer of a static check is always positive or negative and the code is accepted or rejected. It is impossible to break the execution in the middle, as in the dynamic approach. A static checking does not necessarily need any support from a runtime environment. In this sense it is more general than the other. However, the static security policies are usually less precise because all nontrivial security policies are of course undecidable. The user must then accept the fact that some programs would be misjudged which means that some perfectly legal programs are sometimes rejected. The opposite situation, where an illegal code was accepted would be a true disaster and should never happen. In practice we then aim at building *conservative* algorithms which accept only valid programs and reject all invalid and as few valid ones as possible.

All these observations lead to an obvious conclusion: there is no perfect way to enforce a safety policy. The best what we could probably do would be to put the advantages of dynamic and static checking together in a framework which unifies all best features of these two.

	Pros	Cons
Dynamic	- precise security policies - exact verification	- non-trivial - *boom* effect
Static	- does not require the VM - no *boom effect*	- *in/too* sensitive

Figure 2.1: Comparison of static and dynamic techniques

This is exactly what XMS is. On one side modern runtime environments such as the Microsoft .NET offer quite sophisticated dynamic policies. Formal aspects of underlying intermediate language also impose some static policies (like type-safety). At top of that the XMS dynamic verification engine uses code instrumentation to verify the safety policy at the run time.

On the other side, the pros of dynamic security built into these environments can be extended with static policies which would allow the user to be even more confident that the code is perfectly safe in variety of ways. In its current form XMS can statically validate the Design By Contract safety policy, however because it is built on top of the Proof Carrying Code paradigm, it is imaginable that it could validate any safety policy expressible in a formal way.

2.4 Modularization and compositionality

There are other important aspects of safety that have an influence on the way security policies are enforced.

First aspect is **modularization**. Ideally it would be desirable to prove that a big module consisting of many smaller ones is safe by breaking up the safety requirement to these smaller parts of it. We would just like the following property to hold:

$$SAFE(\mathcal{M}) \Longleftrightarrow \bigwedge_{F \in \mathcal{M}} SAFE(F)$$

That is exactly how the Design By Contract safety policy behaves. We do not validate the whole execution graph of the application. Instead, we analyze every single method separately and then conclude that the application is safe regardless of the specific execution trace.

Second important aspect of safety is **compositionality**. In concurrent environment it would be desirable to prove that a concurrent system is safe only and only if all its parallel components are. We would like to have:

$$SAFE(M_1 \mid \ldots \mid M_n) \Longleftrightarrow SAFE(M_1) \wedge \ldots \wedge SAFE(M_n)$$

Although the Design By Contract policy is not compositional "out-of-the-box", it is known that the analysis of the concurrent execution can be performed ([2]) and in the future XMS could probably adopt these techniques.

Nevertheless, these two properties, modularity and compositionality, are not general. They hold for some safety properties but do not hold for others. What is more unfortunate, they are completely independent of each other. For example, another important safety policy, the Non-Interference ([14]) is compositional but not modular. This would bring interesting issues if such policy would be adopted to XMS.

Chapter 3

Core Paradigms

3.1 Design By Contract

3.1.1 Overview of the Paradigm

The **Design By Contract** ([26]) paradigm lays the base for systematic object-oriented development. It defines a precise framework where software components can be seen as communicating entities whose interaction is based on mutual obligations and benefits.

These obligations take the form of predicates which precisely define what requirements must be met for a code to be run and what is the result of the execution. Assuming that there is a code supplier module and code client module, there are several important predicates that must be provided for the DBC:

precondition of a method is a predicate which forms an obligation for the client (the client side has to make the precondition satisfiable upon the invocation of the method) and a benefit for the supplier (it assures the supplier that cases that do not satisfy the precondition have not to be handled)

postcondition of a method is a predicate which forms an obligation for the supplier (the supplier side has to make the postcondition satisfiable upon the method's exit) and a benefit for the client (it precisely defines the state of supplier's computation)

class invariant is a predicate which obligates the supplier to make it satisfiable for all the time the client has an access to an object instance

invariants are predicates used internally by the supplier to describe the state maintained upon each entry into a loop body

Contracts can be seen as a strong mechanism for enforcing a **safety of computation**. For any method to be predictable, if precondition is met when the method is run then we should expect that the execution will end in a state in which the postcondition is also met. Contracts are also a form of **formal documentation** since they provide a detailed description of code semantics.

The Design By Contract paradigm introduces **multi-level testing**. According to this idea Contracts should not be verified only in a production environment but rather during the testing process. There are **unit tests** where modules are tested in isolation and assuming that client obligations are met, tests verify that supplier's obligations are correct. There are then **integration**

tests where modules are tested together and then the primary focus is on verifying the client's obligations.

3.1.2 Contracts in Practice

Contracts are gradually introduced in new programming languages. In some programming languages they are even a first-class constructs (Eiffel [25], Nemerle, D).

For other languages there are usually many DBC implementations to choose from, just to mention a few:

- *DBC for C*[1] and *GNU Nana*[2] are DBC implementations for the C/C++ programming languages
- *eXtensible C#*[3] is a postcompiler that transforms declarative contracts in C# applications into a code that verifies pre- and postconditions of methods and properties
- *ContractForJ*[4], *JContractor*[5] and *SpringContracts*[6] are Design By Contract implementations for the Java and they use various code instrumentation and bytecode augmentation techniques to verify contracts in the run time

Yet another level where Contracts play an important role is the world of software modelling. As models expressed in the **Unified Modelling Language** usually are not precise, their semantics is often expressed using the **Object Constraint Language**, a formal language maintained by the **Object Modelling Group** which is *"used to describe expressions [...] which typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model*". [7] As such, the OCL can be seen as the Design By Contract shifted from the programming language level to the software modelling level and existing OCL frameworks should be also mentioned in DBC context ([3], [9]).

3.2 Proof-Carrying Code

The Proof Carrying Code paradigm was proposed in 1998 ([33]) and is based on earlier works on digital certificates.

3.2.1 Overview

The main idea is a notion of *Verification Condition*, a logic predicate that contains the information about the program execution. The Verification Condition is not just any predicate but rather a special one - if the predicate is provable then the program execution is valid. It turns out that the formal proof of such a predicate can be used as a digital certificate which guarantees that the code is safe according to the safety policy.

[1] `http://dbc.rubyforge.org`
[2] `http://savannah.gnu.org/projects/nana`
[3] `http://www.resolvecorp.com/`
[4] `http://www.contract4j.org/contract4j`
[5] `http://jcontractor.sourceforge.net/`
[6] `http://springcontracts.sourceforge.net/`
[7] OMG Specifications, `http://www.omg.org/technology/documents/modeling_spec_catalog.htm`

The overview of the PCC infrastructure is shown in figure 3.1. The figure depicts the PCC in its simplest form. As we can see there is a kind of a protocol between the Code Producer and Code Consumer. This protocol is sometimes referred to as **PCC protocol**.

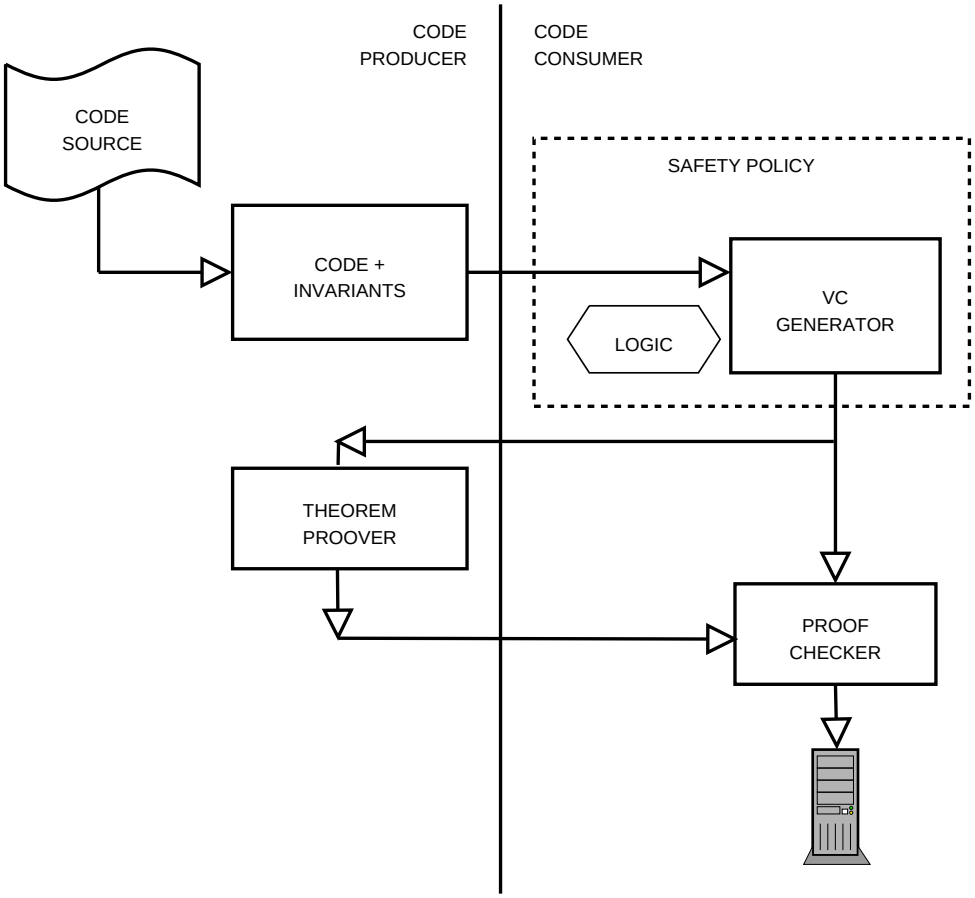

Figure 3.1: Overview of basic PCC protocol

The PCC protocol consists of the following steps:

1. The Safety Policy is defined by the Code Consumer.
2. The Code Producer prepares the source code.
3. The Code Consumer uses the Verification Condition Generator to scan the source code and build a predicate (**Verification Condition**) that is sent back to the Code Producer.
4. The Code Producer is responsible for finding a formal proof of the Verification Condition. The proof of the Verification Condition acts as the certificate of safety.
5. The proof is again sent back to the Code Consumer and validated with a Proof Checker.
6. If the Verification Condition is valid then the code is actually run on a client machine.

Earlier we have discussed several notions of safety/security. No matter what kind of policy we turn into account, it must be somehow expressible formally using some kind of logic

with a precise proof system. This requirement is fundamental for PCC because the Verification Condition built upon the code together with its proof acts as the certificate of the code safety.

Two really difficult issues arise here. These are:

- building a Verification Condition from the code
- concluding that the code is safe if the Verification Condition is provable

In fact for **any** Safety Policy these two issues require a unique *theorem*. In the most general form it can be stated as a *meta*-theorem:

Theorem 3.1 (Meta-theorem of PCC)**.** *For a given Safety Policy S and a code F, if the Verification Condition for the S built from F is valid, i.e.*

$$S \models VC_S(\mathcal{F})$$

then the code F is safe according to S.

This is where the difficulty of PCC lies - for any Safety Policy we need an algorithm for building Verification Conditions (the algorithm is called **Verification Condition Generator** or **VCGen**) and an ”instantiated” version of theorem 3.1 which would formally validate that the VCGen is correct in a sense that safe programs correspond to valid (provable) predicates.

The PCC Protocol presented above is general but not quite practical. It is too complicated to be used efficiently. The main problem is the need for the communication between the Code Producer and Code Consumer. That is why the original protocol was simplified ([33]). The simplification is built upon an observation that the Safety Policy can be shared between Code Producer and Code Consumer. In fact some common policies can be developed, presented to the public and used by all interested parties.

XMS extends this modified protocol with one important detail: all certificates are embedded into the binary’s metadata so the validation engine can verify all certificates at once with no additional resources required. This modified protocol is shown in figure 3.2.

The Code Producer and Code Consumer use the same public and verified safety policy which defines logic and algorithms to build Verification Conditions.

The Code Producer:

1. adds method specifications to the source code,
2. uses VCGen to build and encode Verification Conditions,
3. constructs proofs for VCs,
4. encodes VCs and proofs and binds them into the code

The Code Consumer:

1. uses VCGen to build Verification Conditions,
2. checks if the same VCs have been supplied with the code by the Code Producer,
3. validates the correctness of proofs (certificates).

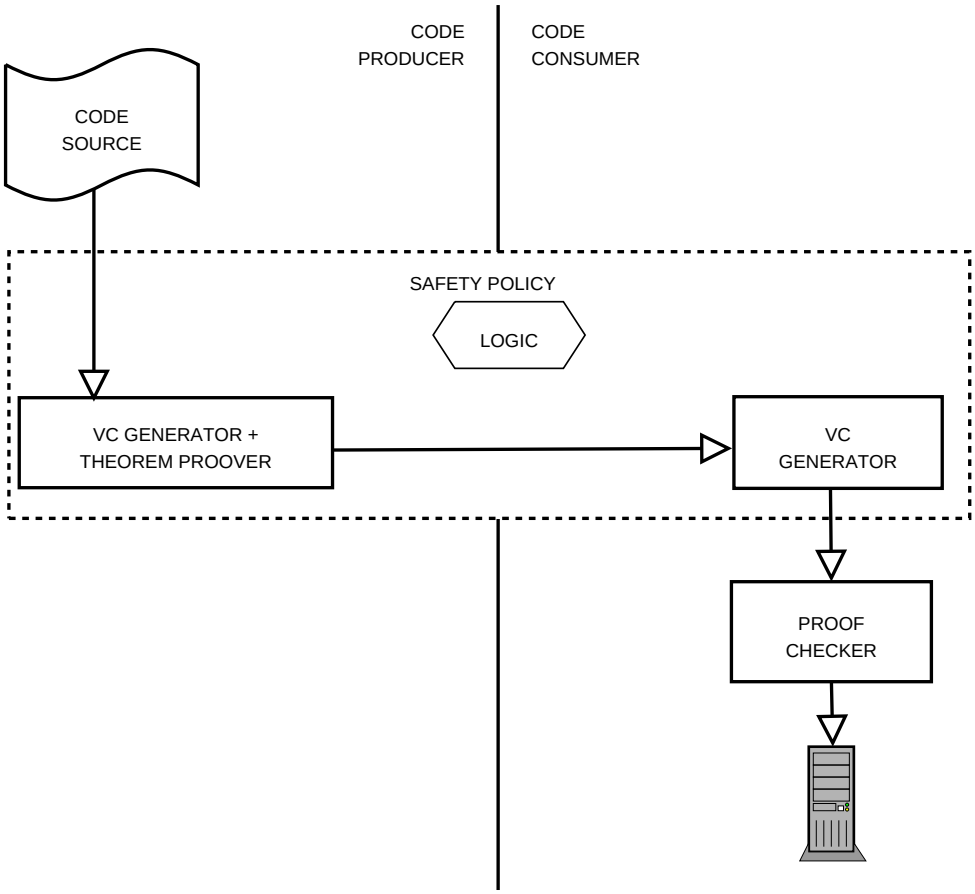

Figure 3.2: A modification of PCC protocol for the XMS

The steps above are required to accept the code as safe. Note that this protocol can fail at several points. Specifically:

1. the code can have no extra information that is required to rebuild the Verification Conditions,
2. the predicates rebuilt at Code Consumer side can differ from these supplied with the code,
3. proofs supplied with the code can be invalid in a sense that they prove something but actual predicates

If the protocol **fails** for any of these reasons the code is **rejected** as unsafe.

Chapter 4

The Intermediate Language

The Microsoft Intermediate Language (MSIL or just IL) is an object-oriented intermediate language. It is executed by the Common Language Runtime (CLR), a runtime environment that itself is a part of Common Language Infrastructure (CLI).

The IL makes it possible to unify software components written in many different programming languages. It supports several high-level language features and that is why it can be easily targeted by most of them. In fact, any language that follows some special rules known as Common Language Specification (CLS) can use MSIL as a back-end, though some languages had to be redesigned. The IL offers class-based objects, inheritance and the type safety, the CLR offers the garbage collection and a rich set of library classes known as the Base Class Library (BCL).

Today a wide set of compilers for a variety of languages is available for the .NET Framework. One of the most widely adopted languages is undoubtedly the C# language designed exclusively for the .NET Platform. As an descendant to C++ and Java, the language is used to develop console-based and window-based application as well as web-based services. The .NET Framework is also a target for a new dialect of Visual Basic, C++ and J# compilers. Compilers are also developed outside Microsoft for Ada, COBOL, Python, Eiffel, Haskell, OCaml, SML and many other languages. Because each higher level compiler targets the same intermediate language, the .NET components written in different languages can cooperate very closely by not only calling each other's methods but also on the level of class inheritance.

In contrast to Java's paradigm:

Java =
One language, any platform.

this new .NET paradigm was concisely summarized by Bruce Eckel ([10]):

.NET =
Any language, one platform.

The full specification of the .NET Runtime Environment as well as the IL and C# languages was published under ECMA standard ([27]). Consequently, the .NET Runtime Environment was also implemented outside Microsoft and released to the Open Source community [1].

[1] Mono and DotGNU projects

4.1 The Runtime Environment

4.1.1 Managed Modules

A *module* is a primary physical building unit of a .NET application. One or several modules can form an *assembly* - a primary logical building unit of an application. Since in majority of practical cases assemblies consist of one module, these two terms - module and assembly - can be used exchangeably.

From the OS perspective there are two types of modules - executables and shared libraries. From the .NET perspective there is almost no difference - the only difference between an executable and a library is the *entry point* - a special marker in the code so the OS recognizes it as the entry point of the application. It is then obvious that a .NET application can consists of a single executable module and an arbitrary number of shared libraries.

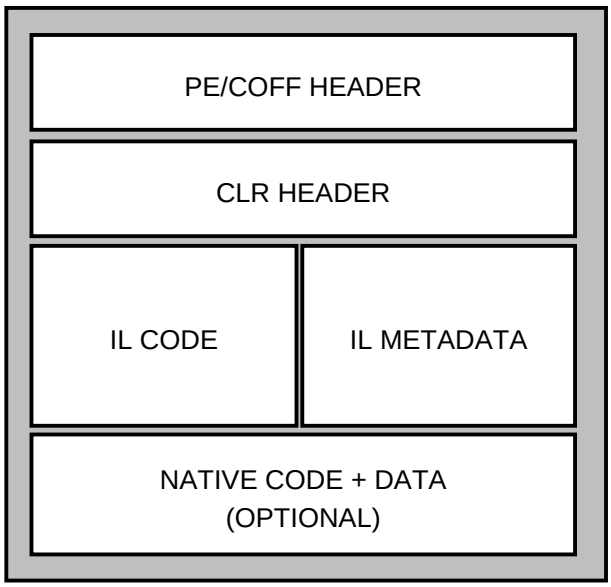

Figure 4.1: Structure of a managed assembly

Figure 4.1 shows a structure of a managed assembly. The first file header is a standard Windows Portable Executable or Common Object File Format (PE/COFF) header. It is followed by the CLR header that describes the structure of the assembly.

The most interesting part of the managed assembly follows. It is called the *metadata*. The module's metadata precisely defines all objects - types, their fields and methods - declared or referenced in the module. It is a complete logical description of the module, data that describes data. Since the metadata is publicly accessible, it is possible to analyze the structure of the module from anywhere in the code. As the metadata is crucial to any interaction between the module and other modules, the integrity of the metadata is precisely checked by the runtime environment before the module is executed.

4.1.2 Execution Process

When MSIL code is executed in the Operating System, it is converted to platform native code by the so called just-in-time (JIT) compiler which is built into the runtime environment. The term ”just-in-time” reflects the fact that rather than compiling all MSIL at once, the JIT compiler converts the code as needed and stores the result for any subsequent calls. Thus, the JIT compilation takes no longer than necessary since when only the part of the code is used in current context, the unused methods will never be JITted. Also the executed code could be, theoretically, even faster then the native code produced by ”static” compilers since the JIT compiler can make use of the information on system architecture to optimize the produced code in current context.

The most important aspect of such execution scheme, however, is the MSIL portability - as the same MSIL can be JITted on almost any architecture (Figure 4.2). For example, if run on a 64-bit platform, JIT compiler can make use of 64-bit registers to perform 64-bit arithmetic as fast as possible. The same code run on 32-bit platform must produce more complex native code to handle 64-bit arithmetic.

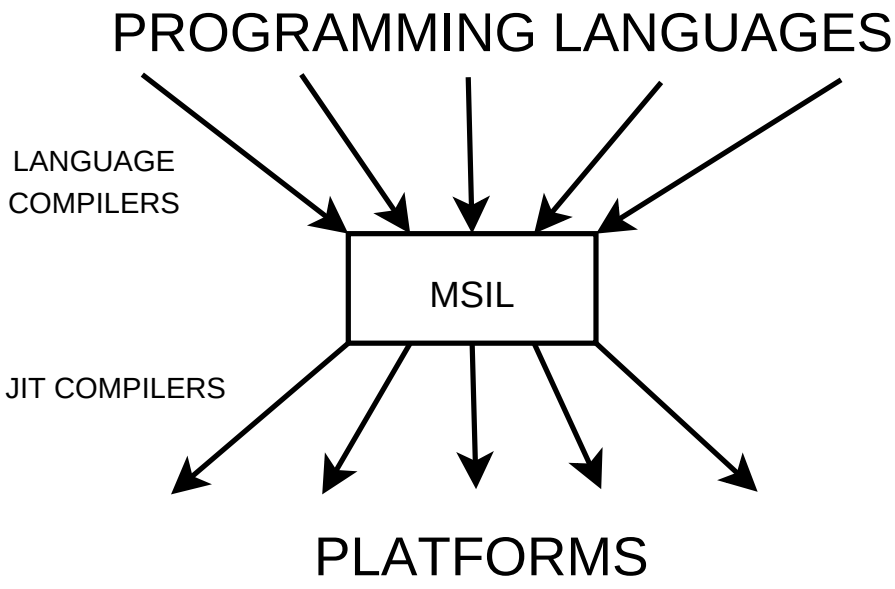

Figure 4.2: High-level compilers versus JIT compilers

4.2 Safety and Security of IL

The Common Language Infrastructure Draft ([27]) defines a strong safety conditions for the Intermediate Language. In fact, the safety is one of the main challenges for the CLI designers. There are also several security mechanisms which can be used by the end-user or by the developer. Before we consider the safety offered by XMS, we have to analyze mechanisms that already have been built into the CLR to see how XMS fits between them.

4.2.1 Safety

Validity and Verifiability

There are several independent levels of safety and security mechanisms in the CLR. As the assemblies are loaded by the runtime environment, a static check is performed to see if the assembly is valid and/or verifiable. If the assembly is not rejected, any other security checks are performed dynamically in the runtime.

There are four types of .NET assemblies: syntactically correct, valid, type-safe and verifiable (Figure 4.3).

An assembly is called *valid* if the file format, the metadata and the IL stored in the assembly are self-consistent. An invalid (inconsistent) IL sequence could be syntactically correct but would for example contain invalid metadata or an opcode that does not belong to the set of valid IL opcodes or a jump to an operand rather than to an instruction.

At the next level of security we have *type-safe* assemblies. Here the ”type-safety” means for example that private methods are not called from outside of a class, i.e. class interaction is based on public information only.

Finally, at the last level of safety there are verifiable assemblies. An assembly is called *verifiable* if it passes a much stronger type-safety check than type-safe assemblies. This test tries to detect any unsafe memory operations that could lead to unauthorized memory access. An unverifiable assembly would for example contain a native code, use pointer arithmetic or convert pointer to values and vice-versa. The verification algorithm is *conservative* - code that is memory safe, can be rejected as unsafe.

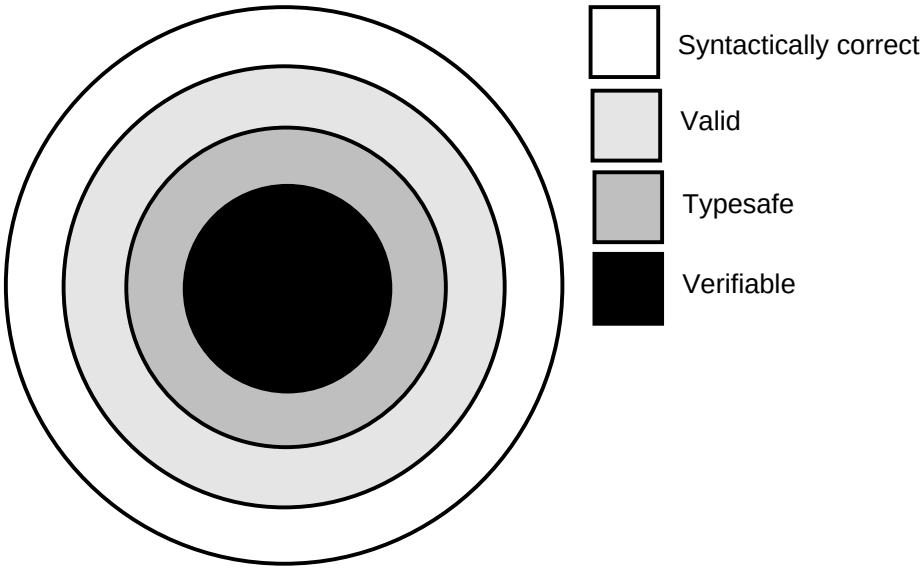

Figure 4.3: Valid and verifiable IL

Below we briefly summarize the type-checking algorithm. The detailed description can be found in part III, section 1.8 of the CLI Draft ([27]).

1. Consecutive IL opcodes are checked for syntactic validity (correct number of parameters).

2. Jump are checked to not to fall out of the method bodies.

3. The one-pass scanner analyzes consecutive opcodes and tries to simulate an execution of the method body. The scanner verifies the stack is properly preserved by emulating the stack state at each opcode. The scanner also verifies that arithmetic and pointer instructions are type-safe:

   - if the opcode pushes the data onto the stack, the type of the data is remembered in the emulated stack
   - if the opcode pops the data from the stack it will succeed only if remembered types are valid in the context of the current opcode
   - the stack depth is remembered at each point so in the case of a backward branch the remembered stack depth at the destination location is compared to actual stack depth (this check is to eliminate the code that could cause stack overflows or stack underflow by continually pushing or popping the data in a loop)
   - if a method is returning a value the stack depth is checked to be 1, if a method is not returning any value the stack depth is checked to be 0 (this is to eliminate a common *buffer underrun* attacks where additional data is pushed onto the stack before the method returns)
   - in case of a method call the returned type is remembered on the emulated stack
   - in case of a type-sensitive arithmetic instruction verifier checks if types of consecutive parameters match the instruction requirements

Because of such a strict algorithm the code is type-safe at MSIL level to quite reasonable extent. Several serious problems faced by some loosely-typed low-level languages are not present in MSIL ([16]). It has also been noticed that the MSIL Runtime Environment had been able to avoid some security issues of Java Runtime Environment ([31]).

This is up to the Runtime Environment to preserve this type-safety at the machine-level when the code is run in the Operating System. In fact, no evidence was found that the existing Runtime Environments break this property in any way (although no formal proof of translation adequacy was given). It means that type-safe code at MSIL level produces type-safe machine code after MSIL is JITted.

The .NET Framework provides an offline verification tool, `PeVerify.exe`. The tool validates and verifies selected assemblies and provides a detailed information about all detected incompatibilities.

Runtime safety

A valid or verifiable assembly executed by the runtime environment is still prone to security issues that can not be detected by a static check. These issues are reported to the application as *exceptions* and can be gracefully handled by the application according to the exception type.

There are several types of security exceptions reported to the application by the CLR, i.a.:

- any reference to an uninitialized object is raised as **NullReferenceException**
- arithmetical MSIL instructions can raise **ArithmeticOverflowException** or **DivideByZeroException**
- array operations can raise **IndexOutOfRangeException**

- any OS security exception is caught and rethrown by CLR, for example an I/O operation on a file to which the current user have no access rights could raise **SecurityException**.

4.2.2 Security

Beside strong safety mechanisms which include static verification of assemblies and runtime detection of unsafe operations, the CLR also gives security mechanisms to both developers and/or end-users.

Assemblies can be executed under three different *hosting environments*:

OS shell - the shell can run assemblies from the command line

ASP.NET - a web application can be hosted inside a web server

Internet Explorer - a web browser can run assemblies referenced by web pages

When an assembly is loaded by the runtime environment, the host environment provides an *evidence* for that assembly. Among any other assembly attributes, the evidence tells the runtime environment where the code comes from. Based on this information, the runtime environment assigns various permissions to the assembly. For example, by default assemblies that are loaded from the local machine are given ”full trust” and have unlimited access to the local file system controlled only by user OS privileges. In contrast, assemblies that are loaded from the Internet zone have no access to the local file system and any file system operation raises a security exception.

Since from the developer’s point of view it is not possible to determine in advance what evidence will be given to an assembly, the same code can run or not according to the context it is executed in. To reduce the possibility of data loss because of unexpected security restrictions, developers can ”secure” the code in two ways, either declaratively, by putting special attributes on methods:

```
[PrincipalPermissionAttribute(SecurityAction.Demand, Role=@"Role1")]
public static void Method() {
  ...
}
```

or imperatively, by calling proper method of a permission object:

```
public static void Method() {

  PrincipalPermission p = new PrincipalPermission( null, "Role1" );
  p.Demand();

  ...
}
```

There are the cases where one of the ways should be used over the other, although in both cases a security exception is thrown either at the method invocation in declarative style or at permission demand in imperative style. The exception can be of course gracefully caught by the application.

From the end-user perspective the **code-access** mechanisms can be used to define custom sets of permissions and custom rules for granting evidences to assemblies, thus creating custom ”sandboxes” for untrusted assemblies. If for example a local assembly can not be fully trusted, a custom sets of permissions can be defined that disallows any interaction between the local file system and the networking subsystem. Then a custom rule can be defined that assigns assemblies from a fixed location, for example the folder `C:\MySandbox`, to be assigned previously defined permission set. If run from local machine, when such ”sandboxed” untrusted application tries to interact with the file system or use the networking system, the security policy would disallow it and a security exception would be thrown.

There are several possible permission types that refer to machine or operating system objects: a file system, the system’s registry, networking system, etc. *Role-based* security attributes can refer to users and their roles in operating system and explicit evidence requests, so that an assembly can be for example run only when it comes from local machine.

The Microsoft .NET Framework contains the **Microsoft .NET Framework Configuration** snap-in (`mscorcfg.msc`), a tool that allows easy and visual manipulation of several code-based security policies. Other CLR implementations can contain similar tools.

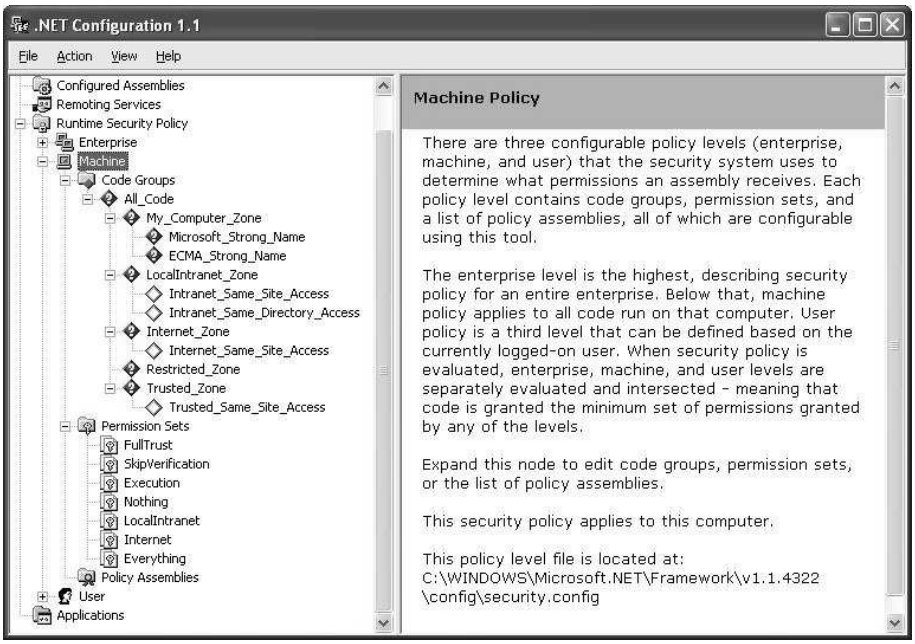

Figure 4.4: .NET Framework Code-based policy configuration tool

Yet another level of security is maintained by the cryptographic signatures which can be used to **sign** assemblies. Any reference to a signed assembly maintain its public key and the integrity of the referenced module can be checked at the runtime. This makes it impossible to replace the signed assembly with an untrusted one.

4.3 The Language

As pointed out in the previous sections, the MSIL together with the Runtime Environment form the core of the .NET Platform.

However, to be able to formally define the semantics of XMS verification engines we have to build the semantics for the language. In subsequent sections we present the IL language and adopt existing description and conventions ([27, 16, 43]) to build the semantics.

Table A contains the full set of IL instructions and comments on their support in XMS.

4.3.1 Naming Conventions

An IL assembly can be seen as a set of types (types are also called ”classes”). A type signature contains fields, methods (including constructors). A signature of a field consists of a name and a type. A signature of a method consists of a name, a type of a returned value and a list of arguments with their types. Methods bodies are sequences of (optionally labelled) instructions. This is summarized in following figure.

M	=	$(C, D, \ldots)$	modules
C, D, T	=	$(f, g, h, \ldots, F, G, H)$	types
f, g			fields
F, G			methods
F_i, G_j			instructions
l			labels

Figure 4.5: Naming conventions

Note that $T :: f$ and $T :: F$ will be used to denote that a particular field f or method F belong to specific class T.

4.3.2 Types (classes)

The IL types fall into two categories: reference types and value types. Reference types represent objects that are stored on the heap and are referenced from the stack or the heap during the computation. Value types are much like C structures, they represent objects that are stored on the stack.

This distinction however is introduced mainly for performance reasons. From the type system point of view, the class hierarchy is single-rooted and the `System.Object` (or just `object`) is a root of the object hierarchy. Even simple value types inherit from the `object` class and can be uniformly treated like regular objects with little performance penalty - boxing and unboxing of value types is handled at the MSIL level.

Type system contains several built-in value types (booleans, integers, etc.) and can be extended with custom value or reference types (figure 4.6).

Types contain fields and methods with specific signatures (figure 4.7). A field signature contains name and type of a field. A method signature contains the name, type of returned value, names and types of arguments and one of 4 **calling conventions**: *static*, *vararg*, *instance* and *instance vararg* (*static* methods do not refer to any specific instance of a class and are opposite of *instance* methods which operate on class instances).

Note that types which are part of the Common Type System have a full name of the form `System.Object`, `System.Int32`, `System.Boolean` etc. indicating that they come from the `System` namespace. Programming languages are free to provide *name aliases* for common types

`void`	no bits
`object`	root of the class hierarchy
`bool`	boolean value
`int32`	signed 32-bit integer
...	other value types
`class` C	custom reference type

Figure 4.6: Types

so that the name alias can be used aside of the type's full name. For example, the MSIL language provides name aliases for common types of the form `object`, `int32`, `bool` etc.

$$\begin{array}{lcll} f & ::= & C \ \text{name} & \text{field signature} \\ F & ::= & \mathit{conv} \ \ C \ \ \text{name} \ (C_1 \, n_1, \ldots, C_k \, n_k) & \text{method signature} \end{array}$$

Figure 4.7: Field and method signatures

4.3.3 Inheritance

The inheritance relation between IL types is presented below using the **subtype** relation. An inherited type contains al least all fields and all methods from its ancestor.

In particular, any type can be used in the context where `System.Object` is expected. We will write $D <: C$ to denote that type D is a subtype (inherits from) type C.

The $<:$ obeys some rules. Two most important are: the class hierarchy is single-rooted and each type inherits from exactly one type. (even value types inherit from the base type `object` however they cannot be inherited).

These rules are summarized in figure 4.8

$$\begin{array}{ll} A <: \texttt{System.Object} & \text{Hi Root} \\ A <: B \wedge B <: C \Rightarrow A <: C & \text{Hi Trans} \\ A <: F \wedge A <: G \Rightarrow F <: G \ \ \vee \ \ G <: F & \text{Hi Single} \\ A <: F \ \wedge \ \text{F is a value type} \Rightarrow A = F & \text{Hi Val} \end{array}$$

Figure 4.8: Basic inheritance rules

4.3.4 Method bodies

IL method bodies are sequences of instructions. It is interesting to notice that many instructions have several forms, most often a *short* and *long* form. Forms differ only in parameter size, a short form requires a 1-byte parameter and a long form requires a 4-byte parameter.

Although the semantics will be presented in following sections, we should notice that the IL is a stack-based language. Every instruction takes its parameters from the top of the stack and puts its result onto the top of the stack. In particular, there is no direct addressing of local

variables or method's parameters except *load* and *store* instructions which move values between the stack and variables / parameters.

The depth of the stack is measured with **slots** not with bytes. For example, when there are two 32-bit integer values and a string value on the stack, the depth is 3. Since the language is strongly-typed, all values contain the information about their type and the incompatibility between actual and expected types are detected during the initial verification phase.

IL instructions fall into 6 categories and in following subsection we present selected instructions.

Control Flow Instructions

Each IL instruction can be labelled by a label. Labels are used to mark the destination points of jumps. For example:

```
  br jump_1
  ...
jump1:
  ...
```

Selected IL control flow instructions are presented in table 4.9.

`br` *label*	unconditional branch
`brtrue` *label*	conditional branch
`brfalse` *label*	conditional branch
`bge` *label*	conditional branch
`bgt` *label*	conditional branch
`ret`	return

Figure 4.9: Selected IL Control Flow Instructions

The `br` *label* instruction unconditionally jumps to an instruction labelled by *label*.

In contrast, the `brtrue` *label* [`brfalse` *label*] instruction pops the value from the stack and jumps to an instruction labelled by *label* only if the popped value is nonzero [zero].

The `bge` *label* and `bgt` *label* instructions pop two values from the stack and jump to an instruction labelled by *label* when the first value is greater-or-equal or greater, respectively.

The `ret` instruction pops the stack frame and ends a method's execution. If a method that have been called should return a value then exactly one value of proper type must be on the stack at the moment the `ret` instruction is invoked. The value is then removed from the stack of the method and is put onto the stack of calling method where it can be then popped and processed.

Arithmetical Instructions

IL supports several arithmetical instructions which are presented in Figure 4.10.

Arithmetical instruction set contains parameterless, unary and binary instructions. Any instruction that needs parameters take them from the stack. The result of an arithmetical instruction is always stored on the stack after the instruction is executed.

`ldc.i4` *i*	load integer value *i* onto the stack
`dup`	duplicate the stack value
`pop`	remove the value from the stack
`add`	addition
`sub`	subtraction
`mul`	multiplication
`div`, `rem`	division, remainder
`neg`	negation
`and`, `or`, `xor`	bitwise AND, OR, XOR
`not`	bitwise unary inversion
`shl`	bitwise shift left
`shr`	bitwise shift right
`ceq`, `cgt`, `clt`	check if first equal to / greater / less than the second

Figure 4.10: IL Arithmetical Instruction Set

Instructions for Addressing Fields, Arguments and Local Variables

Instruction for addressing fields, arguments and local variables are presented in Figure 4.11.

`ldarg` *v*	load from argument
`ldarg.i`	load from *i*-th argument
`starg` *v*	store into argument
`ldloc` *n*	load from local variable
`ldloc.i`	load from *i*-th local variable
`stloc` *n*	store into local variable
`stloc.i`	store into *i*-th local variable
`ldfld` *C T* :: *f*	load from instance field
`ldflda` *C T* :: *f*	load manager pointer from instance field
`stfld` *C T* :: *f*	store into instance field
`ldsfld` *C T* :: *f*	load from static field
`stsfld` *C T* :: *f*	store into static field

Figure 4.11: Selected IL instructions for fields, arguments and local variables

The `ldarg` *v* instruction loads the value of an *v* method argument on the stack. The `starg` *v* instruction takes the value from the stack and stores it in the argument slot *v*.

The `ldloc` *n* instruction loads the value of an *n*-th method's local argument on the stack. The `stloc` *n* instruction takes the value from the stack and stores it in the local variable number *n*.

The `ldfld` *C T* :: *f* instruction pops an object reference *r* from the stack and loads the value of the object's field *T* :: *f* of type *C* on the stack. The `stfld` *C T* :: *f* instruction pops the value from the stack, pops an object reference from the stack and stores the value to the object's field *T* :: *f* of type *C*. The `ldflda` instruction behaves just like `ldlfld` but instead of loading the value of the field on the stack, it loads a reference to the field on the stack.

In contrast, the `ldsfld` *C T* :: *f* instruction loads the value of the static field *T* :: *f* of type

C on the stack. The `stsfld` *C T* :: *f* instruction pops the value from the stack and stores the value to the static field *T* :: *f* of type *C*.

The distinction between these two pairs of instructions (`ldfld`, `stfld` vs `ldsfld`, `stsfld`) is then obvious: they apply to instance and static fields respectively.

Instructions for Calling Methods

IL instructions for calling methods are presented in Figure 4.12

`call` *conv C T* :: *F*	call a method
`callvirt` *conv C T* :: *F*	call a virtual method

Figure 4.12: IL Instructions for Calling Methods

The `call` instruction pops arguments from the stack and call the method with given arguments. For example:

```
ldc.i4      1
call        int32 TheClass::StaticMethod(int32)
```

first loads an integer value 1 onto the stack and then calls a static method which expects single integer parameter and returns an integer value.

Note that the `call` instruction expects a full signature of the method (including the type of a return value) as a parameter. This is to avoid any confusion in case of multiple overloaded methods sharing the same name.

The `callvirt` instruction does almost the same but the call is conducted by an instance's vtable (table of virtual methods) which means that the call is polymorphic.

The distinction between virtual and non-virtual calls is simple: the `call` instruction does not use the vtable which means that it is **never** polymorphic. It is then suited to static functions and non virtual calls. In contrast, the `callvirt` instruction always uses the instance's vtable. If an instance of a class is cast to the parent type and a method is called by `call` then the parent's method will be called but when a method is called by `callvirt` a child method will be called as a result of the vtable entry.

Is interesting is that a non-virtual method can be always safely called with `callvirt` as it will have the same effect as `call`.

Parameters to methods should be pushed on the stack in order of their appearance in the signature (first parameter pushed first, last parameter pushed last). If an instance method is called, the first parameter must always be a reference to the instance of an object (*this* parameter).

Instructions for Addressing Classes and Value Types

As an object-oriented language the IL includes instructions dedicated to classes (table 4.13).

The `ldnull` instruction loads a null object reference on the stack. The `newobj` instruction allocates memory for a new instance of specified class. It pops arguments from the stack, calls appropriate constructor (which must be called *.ctor*) and pushes the reference to a newly created object on the stack.

Example 4.1 Following sequence of code:

`ldnull`	load null reference
`newobj` *T* :: *.ctor*	create a new instance

Figure 4.13: Selected IL Instructions for Class Manipulation

```
ldc.i4      4
newobj      instance void TheClass::.ctor(int32)
```

initializes a new instance of `TheClass` by calling a one-parameter constructor and passing the value 4 to the constructor.

Vector Instructions

The IL also includes few instructions dedicated to vector operations (table 4.14).

`newarr` *token*	create a vector
`ldlen`	get the element count
`ldelem.x`	11 instructions to load vector element
`stelem.x`	8 instructions to store vector element

Figure 4.14: IL Instructions for Vector Operations

The `newarr` instruction pops the element count from the stack and creates the vector of elements specified by the `token`. The `ldelem` and `stelem` groups contain strongly-typed instructions to load and store vector elements. For example, among 11 instructions to load a vector elements we have `ldelem.i4` to load an element of type `int32`, `ldelem.i1` to load en element of type `int8`, `ldelem.u1` to load unsigned element of type `int8` and `ldelem.ref` to load an element of reference type.

Example 4.2 Following sequence of code:

```
ldc.i4 13
newarr System.Int32
```

creates a vector of 13 integer values.

The `ldlen` instruction takes the vector reference from the stack and puts the element count onto the stack.

Each of `ldelem` instructions take the element index and the vector reference from the stack and put the value of the element on the stack. Each of `stelem` instructions take the value to be stored, element index and the vector reference from the stack and put the value of the element into the appropriate slot of the vector. In case of any illegal `ldelem` or `stelem` operation (index of bounds of the vector, null reference for vector, type incompatibility) a runtime exception is thrown.

4.4 The Semantics

The semantics of the IL language is documented in [27], however, it takes a semi-formal form. Because a precise semantics is a core of XMS infrastructure (it has a decisive role in constructions of verification conditions and proofs of their correctness) below we present it in a concise, formal manner inspired by [16] and [43].

4.4.1 Values

The IL language runs under the control of a stack-based machine. Stack size is measured in **slots**, not in bytes. A stack slot can accept one item of any type.

For example, after following sequence of instructions:

```
ldc.i4 0     // put 0 onto the stack
ldc.i4 1     // put 1 onto the stack
ldc.r8 3.14  // put 3.14 onto the stack
ldc.r8 2.71  // put 2.71 onto the stack
```

four stack slots are occupied.

Most instructions take arguments from the stack and place the results back onto the stack. For example following sequence:

```
ldc.i4 1     // put 1 onto the stack
ldc.i4 2     // put 2 onto the stack
ldc.i4 3     // put 3 onto the stack
add          // take top two values from the stack,
             // add then and put result onto the stack
add
```

lefts a single result value 6 on the stack. The result value could be then used as a parameter for another instruction. A **result value** of an instruction can be empty, can be a value type or a reference (Figure 4.15).

u, v ::=	result value
0	void, no result
$i1$, $i2$, $i4$, $i8$	1,2,4 and 8-byte integer
$r4$, $r8$	4 and 8 byte floating-point
p	reference
`null`	null reference

Figure 4.15: Result values

4.4.2 Memory

The Runtime Environment executes the IL code by running a method signed as the *entry point* of the code. Many high-level languages enforce the naming convention on the entry point, for example, in case of the C# language the entry point must be a static, parameterless method called *Main*, optionally accepting a vector of strings as an argument. New stack frames are initialized on the stack as consecutive methods are called.

During the execution of consecutive instructions of a method, the stack frame contains either values or references. For value types, stack contains value of an object and the slot size depends on the size of the value. For reference types, stack contains a reference to the object data, the slot size has the size of a reference and the object data is stored on the heap.

Our semantic model must then contain the stack and the heap. Because classes contain static fields, we need a shared storage where static values from all classes are stored. To complete the description of the method's local memory context we will also need method's local arguments and local variables.

Table 4.16 summarizes the definition of a method's local **memory context**. Since methods call other methods, we will use the notation ρ_F, ρ_G to denote local contexts of different methods. If this is clear from the context or not important to the context, the reference to method's caller will be omited.

ρ_F	::=	$(l_A, l_V, h, H, s, \rho_G)$	local memory context of F
l_A	::=	$[a_0 \mapsto \mathtt{a_0}, \ldots, a_n \mapsto \mathtt{a_n}]$	local arguments
l_V	::=	$[v_0 \mapsto \mathtt{v_0}, \ldots, v_n \mapsto \mathtt{v_n}]$	local variables
h	::=	$p_i \mapsto o_i^{\,i\in 1\ldots n}$	heap
o	::=	$T[f_i \mapsto u_i^{\,i\in 1\ldots n}]$	object
H	::=	$T_j^{\,i\in 1\ldots k}[f_i \mapsto u_i^{\,i\in 1\ldots n}]$	shared storage
s	::=	$u_0, \ldots, u_n$	stack
ρ_G			local memory context of F's caller

Figure 4.16: Local memory context

Note that the entry point method's local context must point to a fixed memory context which is handled by the runtime environment so that the entry point method could return values to the operating system.

Local arguments can be seen as a vector $l_A = (\mathtt{a_0}, \ldots, \mathtt{a_n})$ of values and **local variables** as a vector $l_V = (\mathtt{v_0}, \ldots, \mathtt{v_n})$ of values.

Both vectors can be also seen as partial functions from variables to their values. What we should point is that the IL method arguments are *named* arguments, so we can reference them *by name* or *by number* but local variables are *numbered* arguments (their name is not present in the binary image) so we can reference them only *by number*.

We will adopt this duality and we will write $l_A(v)$ to denote the value of method argument named v and $l_V(n)$ will denote the value of local variable number n. We will also write $l_A[v \mapsto u]$ ($l_V[n \mapsto u]$ respectively) to denote the local arguments (local variables) vector, in which the value v (n) is updated to a new value u. Formally:

$$(l_A[v \mapsto u])(v') = \begin{cases} l_A(v') & v \neq v' \\ u & v = v' \end{cases}$$

$$(l_V[n \mapsto u])(n') = \begin{cases} l_V(n') & n \neq n' \\ u & n = n' \end{cases}$$

The **heap** is a finite set of the object data, each taking the form $T[f_i \mapsto u_i^{i \in 1 \ldots n}]$ where T is the type and the mapping $f_i \mapsto u_i^{i \in 1 \ldots n}$ maps instance fields to result values (for example, a field value can be another reference). The object data can be **referenced** and we will write $p \mapsto o$ to indicate that the reference p references the object data o from the heap.

We will write $p_T(f)$ to denote the value of instance field f from an object of type T referenced by p. We will write $p_T[f \mapsto u]$ to denote an object in which the value of the instance field f has been changed to u. Formally:

$$(p_T[f \mapsto u])(f') = \begin{cases} p_T(f') & f \neq f' \\ u & f = f' \end{cases}$$

Vectors (arrays) are special kind of objects. They map 0-based indexes to actual values and take the form $T[i \mapsto u_i^{i \in 0 \ldots n}]$ to denote a vector of values of type T indexed from 0 to n where $n + 1$ is the length of the vector. A vector is stored in a single slot on a stack.

We will write $a_T(i)$ to denote the value indexed by i from a vector of type T referenced by a. We will write $a_T[i \mapsto u]$ to denote a vector in which the value at the index i has been changed to u. Formally:

$$(a_T[i \mapsto u])(i') = \begin{cases} a_T(i') & i \neq i' \\ u & i = i' \end{cases}$$

We will write $|a|$ to denote the length of vector a. Note that since the length of a vector is set during the initialization, it is fixed and cannot be changed during the lifetime of the vector.

References are result values and they are usually stored in variables. When the value of a variable that holds a reference changes, the object data that is referenced by this reference is lost. The Runtime Environment introduces the Garbage Collector (GC), a mechanism that is independent of the code execution and which sweeps the heap and removes object data that are no longer referenced using generational mark-sweep algorithm. Note that although extremely practical, the Garbage Collector purpose is purely technical - it would be unnecessary if the heap was infinite and new object data could be allocated at demand. Thus, the existence of the Garbage Collector does not affect the language semantics.

The **shared storage** is a finite map from class names to their memory representations, each one takes the form $T[f_i \mapsto u_i^{i \in 1 \ldots n}]$, where T is a class name and the mapping $f_i \mapsto u_i^{i \in 1 \ldots n}$ is a mapping from static field names to result values.

We will write $T(f)$ to denote the value of static field f from class T. We will write $T[f \mapsto u]$ to denote a class in which the value of the static field f has been changed to u. Formally:

$$(T[f \mapsto u])(f') = \begin{cases} T(f') & f \neq f' \\ u & f = f' \end{cases}$$

The **stack** is a vector of result values $(u, \ldots, v)$. We assume that stack grows from right to left, so the (v, s) (or $v \cdot s$) will denote a stack s with v put at the top. We will write $|s|$ to denote the length of the stack s.

Example 4.3 Consider a memory context ρ where

$$\rho \vdash l_A = [x \mapsto 0, y \mapsto 1]$$

and let a predicate P

$$P \stackrel{def}{=} x \geq 0 \ \wedge \ y - x > x$$

We then have

$$\rho(P) = 0 \geq 0 \ \wedge \ 1 > 0 = \mathit{false}$$

which can be read as ”the predicate P does not hold in memory context ρ”.

4.4.3 Instructions

The body of any method F is a vector of instructions. We assume that these instructions are numbered, starting from 0. We write $Dom(F)$ for the set of numbers of all instructions from F. We also write F_i to address the i-th instruction of the method.

We model the execution state as a tuple $\Sigma = (i, \rho)$ that contains a program counter $i \in Dom(F)$ and a local memory context ρ. In a fixed context, we will sometimes write $(i, (l_A, l_V, h, H, s))$ instead of (i, ρ)

The operational semantics is defined as a formal judgement of a form $F \vdash (i, \rho) \mapsto (j, \rho')$. The judgement says that the execution of F takes one step from state (i, ρ) to state (j, ρ').

We assume that $0 \in Dom(F)$ and that the execution of F starts in a state $\Sigma_0 = (0, l_A, l_V, h, H, \epsilon))$ (it means that $0 \in Dom(F)$ for any F) where $l_V(u) = 0$ for any local variable u of value type and $l_V(u) = \mathit{null}$ for any local variable u of reference type.

The semantics of selected instructions from all groups are presented below.

$$\frac{F_i = \texttt{br}\ l}{F \vdash (i, \rho) \mapsto (L_F(l), \rho)} \quad \text{Eval } \texttt{br}$$

$$\frac{F_i = \texttt{brtrue}\ l}{F \vdash (i, \ldots, 0 \cdot s) \mapsto (i+1, \ldots, s)} \quad \text{Eval } \texttt{brtrue0}$$

$$\frac{F_i = \texttt{brtrue}\ l \wedge n \neq 0}{F \vdash (i, \ldots, n \cdot s) \mapsto (L_F(l), \ldots, s)} \quad \text{Eval } \texttt{brtrue}$$

$$\frac{F_i = \texttt{brfalse}\ l}{F \vdash (i, \ldots, 0 \cdot s) \mapsto (L_F(l), \ldots, s)} \quad \text{Eval } \texttt{brfalse0}$$

$$\frac{F_i = \texttt{brfalse}\ l \wedge n \neq 0}{F \vdash (i, \ldots, n \cdot s) \mapsto (i+1, \ldots, s)} \quad \text{Eval } \texttt{brfalse}$$

$$\frac{F_i = \texttt{bge}\ l \wedge u < v}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, s)} \quad \text{Eval } \texttt{bge0}$$

$$\frac{F_i = \texttt{bge}\ l \wedge u \geq v}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (L_F(l), \ldots, s)} \quad \text{Eval } \texttt{bge}$$

$$\frac{F_i = \texttt{bgt}\ l \wedge u \leq v}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, s)} \quad \text{Eval } \texttt{bgt0}$$

$$\frac{F_i = \texttt{bgt}\ l \wedge u > v}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (L_F(l), \ldots, s)} \quad \text{Eval } \texttt{bgt}$$

$$\frac{F_i = \texttt{ret}}{F \vdash (i, \ldots, v \cdot s_F, \rho_G) \mapsto \rho_G[s_G / v \cdot s_G]} \quad \text{Eval } \texttt{ret}$$

Figure 4.17: Semantics of selected Control Flow Instructions

$$\frac{F_i = \texttt{ldc.i4}\ i}{F \vdash (i, \ldots, s) \mapsto (i+1, \ldots, i \cdot s)} \quad \text{Eval } \texttt{ldc}$$

$$\frac{F_i = \texttt{dup}}{F \vdash (i, \ldots, v \cdot s) \mapsto (i+1, \ldots, v \cdot v \cdot s)} \quad \text{Eval } \texttt{dup}$$

$$\frac{F_i = \texttt{pop}}{F \vdash (i, \ldots, v \cdot s) \mapsto (i+1, \ldots, s)} \quad \text{Eval } \texttt{pop}$$

$$\frac{F_i = \texttt{add}}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, u + v \cdot s)} \quad \text{Eval } \texttt{add}$$

$$\frac{F_i = \texttt{sub}}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, u - v \cdot s)} \quad \text{Eval } \texttt{sub}$$

$$\frac{F_i = \texttt{mul}}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, u * v \cdot s)} \quad \text{Eval } \texttt{mul}$$

$$\frac{F_i = \texttt{div}}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, u/v \cdot s)} \quad \text{Eval } \texttt{div}$$

$$\frac{F_i = \texttt{rem}}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, u\ MOD\ v \cdot s)} \quad \text{Eval } \texttt{rem}$$

$$\frac{F_i = \texttt{neg}}{F \vdash (i, \ldots, v \cdot s) \mapsto (i+1, \ldots, -v \cdot s)} \quad \text{Eval } \texttt{neg}$$

$$\frac{F_i = \texttt{and}}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, u\ AND\ v \cdot s)} \quad \text{Eval } \texttt{and}$$

$$\frac{F_i = \texttt{or}}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, u\ OR\ v \cdot s)} \quad \text{Eval } \texttt{or}$$

$$\frac{F_i = \texttt{xor}}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, u\ XOR\ v \cdot s)} \quad \text{Eval } \texttt{xor}$$

Figure 4.18: Semantics of Arithmetical Instructions

$$\frac{F_i = \texttt{ceq}\ l \wedge u = v}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, 1 \cdot s)} \quad \text{Eval } \texttt{ceq1}$$

$$\frac{F_i = \texttt{ceq}\ l \wedge u \neq v}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, 0 \cdot s)} \quad \text{Eval } \texttt{ceq0}$$

$$\frac{F_i = \texttt{cgt}\ l \wedge u < v}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, 1 \cdot s)} \quad \text{Eval } \texttt{cgt1}$$

$$\frac{F_i = \texttt{cgt}\ l \wedge u \geq v}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, 0 \cdot s)} \quad \text{Eval } \texttt{cgt0}$$

$$\frac{F_i = \texttt{clt}\ l \wedge u > v}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, 1 \cdot s)} \quad \text{Eval } \texttt{clt1}$$

$$\frac{F_i = \texttt{clt}\ l \wedge u \leq v}{F \vdash (i, \ldots, u \cdot v \cdot s) \mapsto (i+1, \ldots, 0 \cdot s)} \quad \text{Eval } \texttt{clt0}$$

Figure 4.19: Semantics of Arithmetical Instructions, cont.

$$\frac{F_i = \texttt{ldarg}\ v}{F \vdash (i, \ldots, s) \mapsto (i+1, \ldots, l_A(v) \cdot s)} \quad \text{Eval } \texttt{ldarg}$$

$$\frac{F_i = \texttt{starg}\ v}{F \vdash (i, l_A, \ldots, u \cdot s) \mapsto (i+1, l_A[v \mapsto u], \ldots, s)} \quad \text{Eval } \texttt{starg}$$

$$\frac{F_i = \texttt{ldloc}\ n}{F \vdash (i, \ldots, s) \mapsto (i+1, \ldots, l_V(n) \cdot s)} \quad \text{Eval } \texttt{ldloc}$$

$$\frac{F_i = \texttt{stloc}\ n}{F \vdash (i, \ldots, l_V, \ldots, u \cdot s) \mapsto (i+1, \ldots, l_V[n \mapsto u], \ldots, s)} \quad \text{Eval } \texttt{stloc}$$

$$\frac{F_i = \texttt{ldfld}\ \ C\ \ T :: f}{F \vdash (i, \ldots, p \cdot s) \mapsto (i+1, \ldots, p_T(f) \cdot s)} \quad \text{Eval } \texttt{ldfld}$$

$$\frac{F_i = \texttt{stfld}\ \ C\ \ T :: f}{F \vdash (i, \ldots, v \cdot p \cdot s) \mapsto (i+1, \ldots, p[f \mapsto v], \ldots, s)} \quad \text{Eval } \texttt{stfld}$$

$$\frac{F_i = \texttt{ldsfld}\ \ C\ \ T :: f}{F \vdash (i, \ldots, s) \mapsto (i+1, \ldots, T(f) \cdot s)} \quad \text{Eval } \texttt{ldsfld}$$

$$\frac{F_i = \texttt{stsfld}\ \ C\ \ T :: f}{F \vdash (i, \ldots, T, u \cdot s) \mapsto (i+1, \ldots, T[f \mapsto u], s)} \quad \text{Eval } \texttt{stsfld}$$

Figure 4.20: Semantics of Instructions for Addressing Fields, Arguments and Local Variables

$$\frac{F_i = \texttt{call}\ \ C\ \ T :: G}{\begin{array}{c} F \vdash (i, \ldots, u_n \cdot \ldots \cdot u_0 \cdot s) \mapsto \rho_G \ \ \textit{where } \rho_G \textit{ is a new context} \\ \rho_G = (0, l_A[a_0 \mapsto u_0, \ldots, a_n \mapsto u_n], \ldots, \epsilon, \rho_F) \end{array}} \quad \text{Eval } \texttt{call static}$$

$$\frac{F_i = \texttt{call instance}\ \ C\ \ T :: G}{\begin{array}{c} F \vdash (i, \ldots, u_n \cdot \ldots \cdot u_0 \cdot p) \mapsto \rho_G \ \ \textit{where } \rho_G \textit{ is a new context} \\ \rho_G = (0, l_A[a_{\texttt{this}} \mapsto p, a_0 \mapsto u_0, \ldots, a_n \mapsto u_n], \ldots, \epsilon, \rho_F) \end{array}} \quad \text{Eval } \texttt{call instance}$$

Figure 4.21: Semantics of Instructions for Calling Methods

$$\frac{F_i = \texttt{ldnull}}{F \vdash (i, \ldots, s) \mapsto (i+1, \ldots, \texttt{null} \cdot s)} \quad \text{Eval } \texttt{ldnull}$$

$$\frac{F_i = \texttt{newobj T :: .ctor}(\texttt{u}_0, \ldots \texttt{u}_n)}{F \vdash (i, \ldots, u_n \cdot \ldots \cdot u_0 \cdot s) \mapsto (i+1, \ldots, p \cdot s)} \text{ where } p \mapsto T[f_i \mapsto u_i] \quad \text{Eval } \texttt{newobj}$$

Figure 4.22: Semantics of Instructions for Addressing Objects

$$\frac{F_i = \texttt{newarr T}}{F \vdash (i, \ldots, l \cdot s) \mapsto (i+1, \ldots, a \cdot s)} \text{ where } a \mapsto T[i^{\,i \in 1 \ldots l} \mapsto 0/\text{null}] \quad \text{Eval } \texttt{newarr}$$

$$\frac{F_i = \texttt{ldlen}}{F \vdash (i, \ldots, a \cdot s) \mapsto (i+1, \ldots, l \cdot s)} \text{ where } l = |a| \quad \text{Eval } \texttt{ldlen}$$

$$\frac{F_i = \texttt{ldelem}}{F \vdash (i, \ldots, i \cdot a \cdot s) \mapsto (i+1, \ldots, a(i) \cdot s)} \quad \text{Eval } \texttt{ldelem}$$

$$\frac{F_i = \texttt{stelem}}{F \vdash (i, \ldots, v \cdot i \cdot a \cdot s) \mapsto (i+1, \ldots, a[i \mapsto v] \cdot s)} \quad \text{Eval } \texttt{stelem}$$

Figure 4.23: Semantics of Instructions for Vector Operations

Chapter 5

The Infrastructure

5.1 The Safety Policy

In previous sections we have presented two core paradigms of XMS, the Design By Contract and the Proof-Carrying Code. To formally define PCC for XMS Safety Policy we have to provide:

- formal definition of a Safety Policy
- an algorithm for the Verification Condition Generator accompanied by the Theorem of Soundness, an ”instantiation” of the Theorem 3.1.

This is where the two paradigms are unified within the XMS: the safety policy is expressed on top of DBC. To achieve this, the code producer has to provide a complete specification of each method of the code - method signatures have to be extended with *preconditions, postconditions* and *loop invariants*.

In contrast to the classic DBC we also require one additional part of the extended specification - the list of arguments (or their internal properties) which are modified by the method’s body such that the modification affects the caller side. This is purely technical and could be avoided with the additional scan of the callee method’s body but in the current implementation it speeds up the analysis. In addition - the analogous list must be provided for each loop invariant for same technical reasons.

The intuition behind this requirement is that when a method is called and the call returns, some variables (or their internal properties) passed as arguments to the callee can be modified. For example, when a value type variable is passed by reference (`ref` in C#) it can be potentially modified in the method’s body and the modification affects the caller side. From the other side, any modification to a value type variable which is not passed by reference does not affect the caller. As for loop invariants - a loop body can modify some variables and although the list of modified variables could be also determined by the additional scan of the loop body, knowing it in advance speeds up the analysis.

The reader probably notice that the requirement to provide in an explicit way the list which could be determined otherwise could be a potential security hole in the infrastructure. Nevertheless, this is not the case in XMS - in fact both list are rebuilt but not in advance but rather during the analysis and the analysis fails if rebuilt list differs from the one provided in the specification. This explicit requirement is not a security hole but rather it makes it possible to perform a one-pass analysis.

Eventually, the specification of each method is of the form:

$$Spec_F = (Sig_F, Pre_F, Post_F, Inv_F, Modif_F)$$

where Sig_F is a method's signature, Pre_F is a precondition predicate, $Post_F$ is a postcondition predicate, Inv_F is a partial function that maps instruction numbers to invariants and $Modif_F$ is the list of parameters that are modified by the method's body.

How the specification is physically provided is a technical detail but in our case the specification is stored in the metadata thus being inseparable from the actual code.

Another important feature would be the **class invariants**, predicates which are valid any time the client has an access to the object instance. Note however that the class invariants can be modelled by conjunctions with all preconditions and all postconditions of class methods' specifications.

Example 5.1 For the following method:

```
.method public static int32 Foo( int32 a, int32 b, int32 c ) {

  ldarg 1
  dup
  mul          // b*b

  ldarg 0
  ldarg 2
  ldc.i4 4
  mul          // 4*c
  mul          // ...*a

  sub          // b*b - 4*a*c

  ret
}
```

we could write down following specification:

$$\begin{array}{lll} Sig_{Foo} & := & \texttt{int32 Delta(int32 a, int32 b, int32 c)} \\ Pre_{Foo} & := & \texttt{true} \\ Pos_{Foo} & := & VALUE = b * b - 4 * a * c \\ Inv_{Foo} & := & \emptyset \\ Modif_{Foo} & := & \emptyset \end{array}$$

Following definition formally states the notion of security with regard to Static Contracts Safety Policy.

Definition 5.1 (XMS Safety Policy)**.** *An MSIL method F having the precondition Pre_F and postcondition $Post_F$ is safe with regard to Design By Contract safety policy if for any initial state of the local store $\Sigma_0 = (0, \rho_0)$ such that $\rho_0 \models Pre_F$ and any state $\Sigma = (i, \rho)$ reachable from the initial state we have that if $F_i =$ `ret` then $\rho \models Post_F$. We will denote this fact as $Safe_{SC}(F)$.*

$$Safe_{SC}(F) \Longleftrightarrow \forall_{\Sigma_0=(0,\rho_0),\,\Sigma=(i,\rho)}\ \rho_0 \models Pre_F \ \wedge\ \Sigma_0 \mapsto^* \Sigma \ \wedge\ F_i = \texttt{ret} \Rightarrow \rho \models Post_F$$

A module $\mathcal{M}$ is safe with if all methods from the module are safe. We will denote this fact as $Safe_{SC}(\mathcal{M})$.

$$Safe_{SC}(\mathcal{M}) \Longleftrightarrow \forall_{F\in\mathcal{M}}\, Safe_{SC}(F)$$

Note that precondition and postcondition play a crucial role in the above definition. A method is safe only if whenever its precondition is valid upon invocation then the postcondition is valid when the method is about to return. Because of the way the policy is defined, it is modular (page 10) and even large applications can be successfully verified.

The above definition has two interesting consequences.

First, a method is safe if the precondition is invalid upon the invocation (even though the postcondition can be invalid when the method terminates). In theory, such situation should never happen because, as we will see, a method invocation is guarded in a sense that the its precondition is verified at the invocation time.

Second, if a method does not terminate at all, it is also safe (because there is no chance to falsify the postcondition). It means that the Design By Contract policy does not aim at verifying the total correctness.

Example 5.2 Let us look at the method F which takes positive integer parameter, increments it and returns this new value as the result. Method F could have following specification:

$$\begin{array}{lcl} Sig_F & = & \texttt{int}\ F(\texttt{int}\ x) \\ Pre_F & = & x > 0 \\ Post_F & = & VALUE > x \\ Inv_F & = & \emptyset \end{array}$$

```
.method public hidebysig static int32 F(int32 x) cil managed
{
  .locals init (int32 v_0)

    ldarg  x
    ldc.i4 1
    add
    stloc  0      // v_0 = x+1

    ldloc  0
    ret           // return v_0
  }
```

Let us trace a particular execution of F:

```
ldc.i4   1
call     int32 F(int32)  // F(1)
```

Upon the invocation of F we have $\rho_0(Pre_F) = 1 > 0$ so the precondition of F is satisfied. There is only one exit point from the method, it is the state $(5, \rho)$ and we have that $F_5 = \texttt{ret}$ and $\rho(Post_F) = 2 > 1$.

According to the definition 5.1 this particular execution of F is then safe. As long as we do not consider arithmetic overflows, we could even informally argue that any execution of F is safe.

5.1.1 Specification language

Specification language must be expressive enough to encode as wide range of interesting safety properties as possible. In case of XMS Design By Contract policy, specifications are encoded in standard first-order logic extended with arithmetic operators and relations as well as some object-oriented additions like field/vector access operations.

$$\begin{array}{rcl} Base & ::= & c \mid v \mid a \mid a\_ORIGINAL \mid VALUE \mid THIS \\ Exp & ::= & Base \mid Exp.f \mid Exp[i] \mid Exp.M \mid F(Exp_1, \ldots, Exp_n) \\ Pred & ::= & Exp \mid \neg P \mid G(P_1, \ldots, P_m) \mid \exists z.\ Pred(z) \mid \forall z.\ Pred(z) \end{array}$$

where

- c is a constant value
- v is a local variable name
- a is a local argument name
- *a_ORIGINAL* refers to the original value of an argument (used only in postconditions)
- *VALUE* refers to the value returned from the method (used only in postconditions)
- *THIS* is resolved as the reference to current instance in an instance method
- $Exp.f$ is a value of field f of object denoted by Exp
- $Exp[i]$ is a value from vector Exp at the index i
- $Exp.M$ is a value returned from method M of object denoted by Exp
- F, G are any arithmetic or logic operators / functions recognized and handled by the theorem proving infrastructure

Note also that this definition in fact makes the specification language close to OCL (page 12). Indeed, except these few OCL predicates which are beyond the reach of a static analysis (for example `oclIsTypeOf`, `oclIsNew`, `allInstances`), the significant subset of OCL specification language features is directly expressible in our specification language.

Example 5.3 Following OCL specification:

```
Post :
 if  x > 0
   result = x
 else
   result = -x
```

in XMS is expressed as

$$\Big((x > 0) \implies VALUE = x \Big) \wedge \Big((x \leq 0) \implies VALUE = -x \Big)$$

Invariants

Invariants are predicates that guard computation of loops. In a sense they set a binding of certain variables which are ”fixed” during the loop executions.

However, from the safety point of view it is not enough to ”bind” variables for the loop body. It is because values of some variables can be changed in a loop body while keeping the invariant valid. Because from the static analysis' point of view a loop can be executed arbitrary number of times, some variables can then have an unknown value when a loop terminates (or at least values of such variables cannot be determined statically).

Because the IL is the stack-based language, the loop body can also change stack values and in extreme cases it can refill the stack with completely new values.

Because of these two observations, XMS has a special form of invariants:

$$Inv_F(i) = (P, Var, k), \text{ for any } i \in Dom(F)$$

where P is a invariant predicate, Var is a set of variables that are modified inside the body of a loop (between the invariant is seen for a first and second time) and k is a depth the stack is modified up to in the loop.

Postconditions

Postcondition describe the state of a computation when a method is about to return to the caller. Because the method may return some value to the caller, the first obvious requirement of the postcondition is then to be able to somehow ”use” the returned value in the postcondition.

In our specification language this returned value can be referred as *VALUE*. An example postcondition,

$$VALUE > 0$$

refers to a method in which the returned value of value type is always positive. Another example of a postcondition,

$$VALUE.age > 0 \wedge VALUE.age < person.age$$

refers to a method in which the value of *age* field of returned object should be always between 0 and *person.age*, where *person.age* would probably be the value of *age* field of method's parameter *person*.

From the caller perspective is it then irrelevant if there are several possible return points in the callee because the callee postcondition refers to **any** return point.

0-values

Another requirement of a Design by Contract framework is to be able to specify the original values of method's arguments in postconditions. For example, in the following C# method

```
public void Swap( ref int x, ref int y )
{
  int z = x;
  x = y;
  y = z;
}
```

it is crucial to be able to refer to original values of *x* and *y* in postcondition - as the execution ends the actual value of *x* is equal to original value of *y* and vice versa. In our specification language original values of arguments are referred using `_ORIGINAL` postfix so that in the above example the postcondition would be stated as `x == y_ORIGINAL && y == x_ORIGINAL`.

5.2 Dynamic Verification Engine

5.2.1 Test-Driven Development

Dynamic verification is a process in which the safety policy is verified in the run time of the application. Although dynamic testing is a fundamental principle of the Design By Contract paradigm, much more general development technique has been proposed and is getting more and more interest.

This technique is called the **Test-Driven Development** (TDD, [4], [20]) and is strictly connected with the **Agile** software development methodology ([41], [44]).

The Test-Driven Development requires that so called **Test Suite** is prepared along with the initial version of the code. The Test Suite contains **Test Cases** which call actual methods from the tested code and use explicit assertions which can be used to verify that the implementation of methods being tested is compatible with the provided specification. These test cases can be then conducted automatically and, in case tests fail, the code is refactored until all test pass.

More formally, TDD consists of 3 major steps ([4]):

1. **Create tests** As long as signatures of actual methods are written down, test cases are also created. Since the code being tested is not implemented yet, all newly created tests should fail. It is not obvious how the test cases should be formulated since it requires an understanding of the code requirements. Often then, tests are added later even when the implementation satisfies tests created earlier.

2. **Write code that passes the tests** As long as tests are prepared, the actual code has to be implemented. The implementation should be conceptually correct but since the main goal is to satisfy the tests, it can be inefficient or inelegant.

3. **Refactor** At this stage the code is refactored until it satisfies some requirements. Many important aspects of semi-automatic refactoring and using the design patterns in refactoring are described in [15]. This phase can be repeated arbitrary number of times until the result is efficient or elegant enough. Often, some advanced design concepts (**Design Patterns**) are stated as a goal of refactoring ([11], [21]).

The TDD methodology can ease the software development and make the software more reliable. By focusing on the specification the developer sees the system with the eyes of its potential client.

This benefit is also a biggest limitation of the TDD. When tests are incorrect or not representative, not only the incorrect code can be written but also many important features can remain unimplemented. Another limitation is the difficulty to use the TDD in complex scenarios where several components must be tested in the same time (for example, the user interface, a remote web service, a database management system) - in such cases it is sometimes impossible to write test cases at all.

And probably the biggest limitation - even the large number of passed tests does not prove that the code is correct. The test case data could be simply not representative enough.

5.2.2 Instrumentation of .NET Code

Although the Test-Driven Development methodology appears to be very useful, it assumes that representative test cases are written and automated tests are performed at the supplier side.

Why shall we do not just put the specification **into** the code so that the preconditions, invariants and postconditions are always **verified** during the code execution?

Well, some implementations of the Design By Contract principle behave exactly this way. At the beginning of method's execution the precondition is explicitly verified and before every exit point the postcondition is explicitly verified. An example could be the C's `assert` macro:

```
int Square( int x )
{
  // explicit precondition verification
  assert( x > 0 );

  int ret = x * x;

  // explicit postcondition verification
  assert( ret = x*x );

  return ret;
}
```

Such approach has one major disadvantage - the specification is too tightly bound to the code. In fact, it is a part of the actual code. When the code changes and the specification does not change (which is quite common), all the places where the specification is explicitly verified also have to be reviewed. This could be extremely painful when the implementation is complicated and there are many different places in the code where such `asserts` appear.

Another approach would be then to use one of so called **Code Instrumentation** techniques. Code Instrumentation assumes that the execution of the code can be somehow altered or supervised in a transparent way and some additional functionality can be "injected" during this modified execution.

From the Design By Contract's point of view we are interested in intercepting methods' invocations and exits. This can be seen as one of so called **concerns** of the **Aspect Oriented Programming** paradigm ([22]) and use one of the existing .NET AOP Frameworks ([40]) or adopt one of two possibilities:

.NET Profiler API ([3]) an external API for providing a profiler functionality on the .NET platform. The profiler makes it possible to put custom code into the execution pipeline,

however it works at the IL stream level (sequence of bytes which have to be decompiled) and the API is not well documented.

Context-Bound Objects this route is taken by the XMS

The documentation defines Context-Bound Objects as[1] *"Objects that reside in a context and are bound to the context rules are called context-bound objects. A context is a set of properties or usage rules that define an environment where a collection of objects resides. The rules are enforced when the objects are entering or leaving a context. Objects that are not context-bound are called agile objects."*

The definition is rather obscure, however the idea is perfectly suited for code instrumentation. Inheriting from the `ContextBoundObject` class makes it possible to define rules used by the CLR for objects for which the execution context must be traced. In case of the XMS we have to put the custom code into the invocation and exit pipeline.

Following steps are necessary to inject custom code into the execution pipeline:

1. a **context attribute** must be declared on intercepted class. Context attributes are just attributes but they define a new execution contexts for context bound objects. In our case this new context is defined inside the `InterceptProperty` class

```
[AttributeUsage(AttributeTargets.Class)]
public class XMSInterceptAttribute : ContextAttribute
{
  ...
  public override void
  GetPropertiesForNewContext( IConstructionCallMessage ctorMsg)
  {
    ctorMsg.ContextProperties.Add( new InterceptProperty() );
  }
  ...
}
```

   The context attribute is then used like any attribute to provide runtime-available metadata for a class:

```
[XMSInterceptAttribute()]
public class ClassWithInterceptedMethods
{
  ...
}
```

2. So called **message sinks** provide a built-in[2] mechanism for providing custom logic into the invocation pipeline of context bound objects. The `InterceptProperty` context property inserts a new message sink into the invocation chain:

[1] `http://msdn2.microsoft.com/en-us/library/system.contextboundobject(VS.71).aspx`
[2] `http://msdn2.microsoft.com/en-us/library/c2k19cxd(VS.71).aspx`

```
public class InterceptProperty :
  IContextProperty,
  IContributeServerContextSink
{
    ...
    #region IContributeServerContextSink Members
    public IMessageSink
      GetServerContextSink(IMessageSink nextSink)
    {
      return new InterceptSink( nextSink );
    }
    #endregion
    ...
}
```

3. the XMS's message sink, `InterceptSink`, is responsible for preprocessing - where preconditions are verified - and postprocessing - where postconditions are verified. Since original values of method's parameters can be used in postconditions, these original values must be stored during the preprocessing phase so that they are available during the postprocessing.

```
public class InterceptSink : IMessageSink
{
  private IMessageSink nextSink;

  public InterceptSink( IMessageSink nextSink )
  {
    this.nextSink = nextSink;
  }

  #region IMessageSink Members
  public IMessage SyncProcessMessage( IMessage msg )
  {
    // preprocess
    IMethodCallMessage mcm      = msg as IMethodCallMessage;
    InterceptContext preContext = new InterceptContext( mcm );
    bool pre =
      this.PreProcess(
        preContext /* use actual values during postprocessing */
        );

    // actual call
    IMessage                retm = nextSink.SyncProcessMessage(msg);
    IMethodReturnMessage     mrm = (retm as IMethodReturnMessage);
    InterceptContext postContext =
      new InterceptContext( msg as IMethodCallMessage );

    // postprocess
    bool post =
```

```
    this.PostProcess(
      preContext,     /* use original values during postprocessing */
      postContext,    /* use actual values during postprocessing */
      mrm.ReturnValue );

  if ( !pre || !post )
  {
    Trace.WriteLine(
      string.Format(
        "Testing failed for {0}.",
        mcm.MethodName ) );
  }

  return mrm;
}

private bool PreProcess( InterceptContext preContext )
{
  ProcessAttribute[] attrs =
    (ProcessAttribute[])preContext.MethodBase
      .GetCustomAttributes(typeof(ProcessAttribute), true);

  bool pre = true;
  for(int i=0;i<attrs.Length;i++)
    pre &= attrs[i].Processor.PreProcess( preContext );

  return pre;
}

private bool PostProcess(
  InterceptContext preContext,
  InterceptContext postContext,
  object retValue )
{
  ProcessAttribute[] attrs =
    (ProcessAttribute[])postContext.MethodBase
      .GetCustomAttributes(typeof(ProcessAttribute),true);

  bool pre = true;
  for(int i=0;i<attrs.Length;i++)
    pre &=
      attrs[i].Processor.PostProcess(
        preContext,
        postContext,
        retValue );

  return pre;
}
```

```
  ...
}
```

In the above code both `PreProcess` and `PostProcess` methods first search for specific **processors** and then use them for pre- and postprocessing. Because of that, several independent processors can be defined for a single method. One processor can just trace the invocation and write some information to a log, another can be used to authenticate the user who is running the application and yet another can be used to verify preconditions and postconditions:

```
public interface IProcessor
{
  // preprocess message
  // varValues: contains method call parameters
  bool PreProcess( InterceptContext varValues );
  // preprocess message
  // varValuesOrg : contains method call parameters
  // varValues    : contains exit parameter values
  // retValue     : contains method's return value
  bool PostProcess(
    InterceptContext varValuesOrg,
    InterceptContext varValues,
    object retValue );
}
```

4. Provide specific functionality of message processors. An example processor which just logs the invocation would be:

```
public class TraceProcessor : IProcessor
{
  public TraceProcessor() {}

  public bool PreProcess( InterceptContext varValues )
  {
    Trace.Write( varValues.MethodName );

    Trace.Write( "( " );
    foreach ( DictionaryEntry de in varValues.Values )
      Trace.Write( string.Format( "{0} {1}, ",
        de.Value.GetType().Name,
        de.Key ) );
    Trace.Write( ")" );

    Trace.Write( " [ " );
    foreach ( DictionaryEntry de in varValues.Values )
      Trace.Write( string.Format( "{0}={1}, ",
        de.Key,
        de.Value ) );
```

```
    Trace.Write( "]" );
    Trace.WriteLine(string.Empty);

    return true;
  }

  public bool PostProcess(
    InterceptContext varValuesOrg,
    InterceptContext varValues,
    object retValue )
  {
    Trace.WriteLine(
      string.Format(
        "retval [{0}]: {1}",
          retValue.GetType(),
          retValue ) );

    return true;
  }
}
```

The most interesting would be of course the XMS processor which is responsible for validating method's pre- and postconditions. In current implementation, XMS processor uses dynamic code generation techniques to build methods that validate substituted predicates by executing them as the C# code. This technique has some minor shortcomings since few logical operators are not directly interpreted by the C# compiler and an additional phase is required where all such operators are converted to corresponding C# constructs.

For example, following specification predicate:

$$\Big((x > 0) \implies VALUE = x \Big) \wedge \Big((x \leq 0) \implies VALUE = -x \Big)$$

would be converted to a C# code:

```
if ( x > 0 )
  VALUE == x;
else
  VALUE == -x;
```

and executed with x and *VALUE* substituted by actual values.

```
public class XMSProcessor : IProcessor
{
  public XMSProcessor() {}

  #region IProcessor Members

  public bool PreProcess(
```

```
    InterceptContext varValues )
  {
    // find the XMS specification
    XMS_Spec[] specs =
      (XMS_Spec[])varValues.MethodBase
        .GetCustomAttributes( typeof(XMS_Spec), true );
    if ( specs != null )
      foreach ( XMS_Spec spec in specs )
      {
        SymbExpr exp = SymbExpr.Parse( spec.Precondition );
        return EvaluateExpression( exp, varValues, varValues, null );
      }

      return true;
  }

  public bool PostProcess(
    InterceptContext varValuesOrg,
    InterceptContext varValues,
    object retValue )
  {
    XMS_Spec[] specs =
      (XMS_Spec[])varValues.MethodBase
        .GetCustomAttributes( typeof(XMS_Spec), true );
    if ( specs != null )
      foreach ( XMS_Spec spec in specs )
      {
        SymbExpr exp = SymbExpr.Parse( spec.Postcondition );
        return EvaluateExpression( exp, varValuesOrg, varValues, retValue );
      }
    }

    return true;
  }
  #endregion

  private bool EvaluateExpression(
    SymbExpr Predicate,
    InterceptContext varValuesOrg,
    InterceptContext varValues,
    object retValue )
  {
    // use dynamic code generation technique
    // to validate the predicate
    ...
  }
}
```

5.2.3 Using the Engine

To use the XMS Dynamic Verification Engine for methods from class *C*, following requirements must be fulfilled:

- determine the class *B* from the class hierarchy that inherits directly from the `System.Object` class and *C* inherits from *B*
- move the *B* class in the class hierarchy so that it inherits from `ContextBoundObject` instead of `System.Object`
- put the XMS context attribute in the definition of *C*

The integration is not perfectly transparent as one could expect but still the specification is external to the code.

Our experiments reveal that the execution of ContextBoundObject for which the XMS specification is verified in the run time is about 500-1000 times slower than the normal execution. The dynamic predicate evaluator which uses code generation is the main bottleneck here. The Dynamic Verification Engine is then not quite usable at the client side since the runtime penalty seems huge but it is perfectly suited for the producer-side testing.

Note, that the engine is not optimized since speed was not a major issue. A highly efficient implementation of dynamic verification engine should probably be built using low-level .NET Profiler API and more efficient predicate evaluator.

Example 5.4 Consider following simple C# code:

```
[XMSIntercept]
public class Test : ContextBoundObject
{
  [Process(typeof(XMSProcessor))]
  [XMS_Spec( "true", "x == y_ORIGINAL && y == x_ORIGINAL" )]
  public void Swap( ref int x, ref int y )
  {
    int z = x;
    x = y;
    y = z;
  }
```

The specification would be:

$$Pre_F = \texttt{true} \qquad Post_F = x == y_0 \ \wedge \ y == x_0$$

Note how the `XMSIntercept` and `Process` attributes are placed on the class and method respectively.

Actual client code:

```
int u = 0, v = 1;
t.Swap( ref u, ref v );
```

The XMS Dynamic Verification Engine outputs:

```
Preprocessing Test.Swap.
Specification found:
Pre=[true]
Post=[x == y_0 && y == x_0]
Precondition : true
Substituted expression : true
Evaluated expression : True
Postcondition : x == y_0 && y == x_0
Substituted expression : 1 == 1 && 0 == 0
Evaluated expression : True
```

5.3 Static Verification Engine

5.3.1 Symbolic Evaluation

According to the Meta-Theorem (3.1) the Verification Condition Generator (VCGen) is the main component of the PCC infrastructure. The original paper ([33]) proposes a pattern called the **Symbolic Evaluator**. The Symbolic Evaluator is a recursive state transformation function that scans the code and produces the Verification Condition (or a significant part of it). We will refer to the Symbolic Evaluator as *SE* and to Verification Condition as *VC*.

Symbolic Evaluation is the process in which a ”virtual” execution is performed but instead of actual values their symbolic representations are used. Whenever the evaluation cannot determine the symbolic result of a computation, a new fresh symbolic value is used.

Symbolic Evaluation of the IL language arise several issues because IL is an object-oriented language. Below we briefly examine these issues:

arithmetics an arithmetic instruction causes VCGen to update its symbolic store to new state.

conditionals a conditional jump causes VCGen to split the symbolic evaluation into recursive paths for all branches. Conditions become assumptions inside the verification predicate.

backward jumps backward jumps could lead to infinite recursion. VCGen requires then that each backward jump targets an instruction for which an *invariant* is provided. Invariants are validated when they are seen for the first time and then validated again when a backward jump is encountered.

method calls a method call puts the method’s precondition as an assumption into the predicate and initializes a new state with all variables which could be modified inside the called method (out parameters) set to new, fresh values.

objects objects are evaluated symbolically in a similar way as in loosely-typed object-oriented languages - by maintaining an internal dictionary which maps field names to their actual symbolic values.

vectors a vector is stored as a index-value dictionary where both indexes and values are symbolic.

polymorphism is it not known until the run-time which exact method is called from a class hierarchy. VCGen relies here on a *subcontracting* principle ([25]) according to which contracts of inherited methods must depend on contracts of base-class methods

0-**values** contracts must allow to use original values in postconditions.

exceptions inside a ”try-catch-finally” block any checked arithmetic instruction, vector reference instructions and method calls could cause an exception to be thrown. All such instructions must be then treated as potential branching instructions.

The Evaluator assumes that the code is verifiable (page 20) which means that there's no possibility of reaching states which are not type-safe. Because of this assumption, there's no need to constantly check whether symoblic states are valid. The evaluation would of course fail on unverifiable code. The Evaluator is presented in Figure 5.1.

The Evaluator (SE) is defined as a recursive function that takes four parameters, written as

$$SE_F(i, \sigma, \mathcal{L}, b)$$

where

- F is a method whose body is evaluated
- i is an address of evaluator's current instruction
- σ is a *symbolic store*
- $\mathcal{L}$ is a loop stack
- b is a boolean variable which controls the way the invariants are handled

A symbolic local store, σ is of the form (l_A, l_V, h, H, s) defined just like its semantic counterpart but values are represented in a symbolic way. Because of the way the invariants are associated with instructions (they do not form separate instructions but rather point to actual instructions), b determines whether an invariant should be taken into account when the instruction is scanned. If b is omitted at all, the default value is `true`. A loop stack $\mathcal{L}$ captures changes to stack and variables during execution of a loop and is described in next subsection.

The symbolic evaluator is run against all methods in a module $\mathcal{M}$ and the global verification condition is build using the resulting predicates. The definition of the Verification Condition is presented in Figure 5.1.

Few notes here. The verification condition for the module is a conjunction of verification conditions for all methods from the module. The verification condition for a method is a conjunction of two predicates, *VCI* and *VCE* named from **V**erification **C**ondition for **I**nheritance and **V**erification **C**ondition for Symbolic **E**valuation.

The *VCI* part of the verification condition is responsible for handling virtual methods and checks if the so called **subcontracting** holds for the method F. If F is virtual then its contract must be compatible with the contract of the same method but from the base class (denoted here as $Base(F)$. Details are presented on page 60.

The *VCE* part of the verification condition is responsible for handling the symbolic evaluation and indeed it starts in first state which assumes that the precondition holds and evaluates the method symbolically starting from the initial state.

Note that the local variables are zeroed in the initial state ($v_i \mapsto 0$)and the local arguments are initialized with the symbolic representations of themselves ($a_i \mapsto a_i$). This initial assignment is correct with respect to the IL semantics, where the local variables are always zeroed and

$$\begin{aligned} VC(\mathcal{M}) &= \bigwedge_{F\in\mathcal{M}} VC(F) \\ VC(F) &= VCI(F) \ \wedge \ VCE(F) \end{aligned}$$

where:

$$\begin{aligned} VCI(F) &= IsVirtual(F) \implies ((Pre_{Base(F)} \implies Pre_F) \ \wedge \ (Post_F \implies Post_{Base(F)})) \\ VCE(F) &= \forall a_0, \ldots, a_n.\, \sigma_0^F \models Pre_F \Rightarrow SE(0, \sigma_0^F, \emptyset, \texttt{true}) \\ \sigma_0^F &= (l_A[a_i \mapsto a_i], l_A[a_i\_ORIGINAL \mapsto a_i, l_V[v_i \mapsto 0], \epsilon) \end{aligned}$$

Figure 5.1: Definition of Verification Condition for Design By Contract

arguments can take arbitrary values upon the method's invocation (these values are set by the caller).

Additionally, a set auxiliary variables referring to original values of arguments is initialized with symbolic representations of these arguments ($a_i\_ORIGINAL \mapsto a_i$). This is to ensure that during the symbolic evaluation of the postcondition (evaluation of the `ret` opcode), original values are available to the evaluator.

5.3.2 Symbolic Evaluation Cases

During the scan, the Symbolic Evaluator simulates the method's execution by updating the symbolic store with respect to current instruction. SE also performs some checks. If any of the checks fail, the code is automatically rejected. In addition, some maintenance actions are performed for some actions.

The essential part of the Evaluator is presented in Table 5.1 on page 62.

From the Evaluator's perspective there are two types of instructions. For some instructions SE does not produce anything, it just changes the state of the symbolic store. For other instructions SE not only changes the state of the symbolic store but also produces a part of the Verification Condition.

When a Symbolic Evaluator scans the method's body, it scans consecutive instructions starting from the first one. If a conditional branch is scanned, the Symbolic Evaluator splits into two copies that scan both branches independently. This of course could lead to infinite recursion in case of backward jumps. That is where invariants play their roles. Invariants help Symbolic Evaluators to avoid the infinite recursion by *guarding* instructions that are targets for backward branches. In fact, no instruction can be a target of a backward jump if it does not have its own invariant. If this is the case - the evaluation fails.

The Symbolic Evaluator will use the information provided in invariants to properly universally quantify variables for loops. For example if variable v_0 is on the list of variables modified during a loop then the VCGen will put $\forall\, v_0$ in appropriate position inside the Verification Condition to indicate that v_0 can have arbitrary value during the loop execution. The same rule applies to stack - arbitrary values can be pushed onto and popped from the stack in a loop body. VCGen will handle this possibility by putting universal quantifiers for all stack values that are modified in a loop body.

A non-empty invariant changes the way the instruction is evaluated. At first the evaluator checks if the invariant is seen for the first time.

When the invariant is seen for the first time, the SE checks if the invariant is valid with

respect to the current state of symbolic store by appending it to the Verification Condition. This corresponds to the fact that the invariant should be valid when a loop is entered. Then the symbolic store is filled with new, fresh values for all variables and stack values that are modified during the loop, the loop invariant is used as the premise in the Verification Condition and the execution continues. This corresponds to the fact that although actual values of variables modified in a loop body are not known in any other but first execution, the invariant still holds. Additionally, a new invariant slot is initialized on top of loop stack $\mathcal{L}$. This new slot, referred as $\mathcal{L}_{TOP}$, is a vector (i, σ, V, s), where i is a number of invariant instruction, σ is a symbolic state in this instruction, V is an initially empty set that accumulates changes to variables, s is a value responsible for tracking the depth up to which the stack is modified inside a loop.

When the invariant is seen for the second time, the SE checks if the invariant is valid after the loop is executed once. The symbolic evaluation of the loop ends here, because if the invariant is valid when the loop is entered and after the loop is executed once then SE concludes by induction that the invariant is valid after the loop is executed an arbitrary number of times. Additionally, an invariant slot that corresponds to the invariant instruction number together with all other slots that were allocated later (nested loops), gives precise information on which variables have been changed in a loop body and what depth the stack has been changed up to. This is where the list of variables which were declared in the modification list of the invariant is compared to the actual list of modified variables. Also the stack is controlled to be within declared limit of modified slots. If any of these checks fail, the evaluation fails.

Formally, we will allow two operations on invariant slots in $\mathcal{L}$ during the execution of SE to track modifications of variables and altering the depth of the stack.

We can add a variable u to the set of modified variables in slots of L. We will denote this operation as $\mathcal{L}(V \leftarrow \{u\})$. We can also trace the stack depth and we will write $\mathcal{L}(s \leftarrow k)$ to indicate that evaluation stack depth increased by k slots. We can also get a set of all variables that were modified in an invariant slot (denoted by $\mathcal{L}(i)$) or get a depth the stack was modified up to in an invariant slot (denoted by $|\mathcal{L}(i)|$).

Control Flow Instructions

Having argued about backward jumps and invariants we can now assume that branching instructions are forward jumps only.

In case of the `br` instruction SE continues the evaluation at the jump destination. The case of the `brtrue`, `bge` and `bgt` instructions is similar but now both branches must be considered in the resulting evaluation. If any of these jump instructions results in a backward jump, SE also checks if the invariant function is defined for the destination instruction. Note that both branches correspond to branching conditions which are put as assumptions in the Verification Condition.

Branching requires that the symbolic evaluator splits into two independent contexts. Technically, the current context is **cloned** and it must be done with proper caution. On one hand, both contexts are independent and changes of values made in one of them must not be visible to the other. On the other hand, during the symbolic evaluation the same reference can be stored in several places (for example a reference can be created on the stack and then stored in two local variables) and the cloning must detect these duplicated references and make only a single copy of it but still duplicated in all places where the reference exists.

The symbolic evaluation of the `ret` instruction depends on the method's signature. If the

evaluated method should not return any value, the VC checks if the method's postcondition is valid with respect to current symbolic store. If the evaluated method should return a value, the VC checks if the method's postcondition is valid with respect to the current symbolic store and the stack value that represent the method's result.

Arithmetical Instructions

In case of `ldc` instruction SE puts the integer parameter at the top of the symbolic stack. In case of `dup` instruction SE duplicates the value at the top of the symbolic stack. In case of `pop` instruction SE removes the value from the top of the stack.

In case of all arithmetic instructions (`add`, `sub`, `mul`, ...) SE performs the symbolic evaluation and puts the result back to the symbolic stack.

Instructions for addressing fields, arguments and local variables

The `ldarg`, `ldloc` and `ldsfld` instructions put the value from local argument, local store or the shared store (respectively) at the top of the stack.

The `starg`, `stloc` and `stsfld` update the local argument, local store or the shared store (respectively) with the value from the top of the symbolic stack.

Instructions For Calling Methods

In case of `call` and `callvirt` instruction VC first checks if the invariant of the callee is valid in current state and then, assuming that the return value is unknown it is set to fresh symbolic value and the postcondition of the callee is valid, SE continues the evaluation at the next instruction. Of course the callee function may not terminate at all but this would be safe according to the definition of this Safety Policy (5.1).

Instructions for Addressing Classes and Value Types

The `newobj` a bit complicated because of the way it is handled by the CLR. The newly created object instance is put onto the stack but is not yet stored into any local variable or method's argument. Symbolic evaluator creates a **symbolic object**, a symbolic representation of a value of reference type and stores it in an auxiliary collection of references. The symbolic object maintains the internal dictionary where field instances are stored. In case of `stlfd` instruction the value from the stack is put into the internal dictionary. In case of the `ldfld` instruction the value from the dictionary is put on the stack **or** there is no value stored in the dictionary yet and the new, fresh value is created and put in the dictionary and onto the stack.

Vector Instructions

The `newarr` instruction raises similar issues as the `newobj` instruction - the newly created vector reference must be stored in yet another auxiliary collection. In this case **symbolic vector** is created and it is responsible for storing the information about symbolic content of the vector.

There are three possible cases for the `ldelem` instruction.

The easiest case is when there exists a value stored in the symbolic vector at the symbolic index which is symbolically equal to the index expression of the `ldelem` instruction. In such case the symbolic value from the symbolic vector is put onto the stack.

It is however possible that the indexing expression is not directly equal to any symbolic indexes stored in the symbolic vector. In such case, a sequence of branching instructions is generated, each one taking the equality of symbolic indexes as the premise.

The last case is when the symbolic index is not equal to any symbolic indexes previously stored in the symbolic vector. In such case a symbolic representation of the symbolic value is put onto the stack.

Example 5.5 Consider following C# code:

```
int Foo( int[] TheArray )
{
  ...
  TheArray[k] = 1;
  TheArray[n] = 2;
  ...
  int v = TheArray[n];
  ...
}
```

What is the symbolic value of v? According to the description above, the simplest case would be to find the symbolic expression stored in the vector under the same symbolic index, in this case n. In the above example there is a value stored at this index and it is 2.

Consider however following code:

```
int Foo( int[] TheArray )
{
  ...
  TheArray[k] = 1;
  TheArray[l] = 2;
  ...
  int v = TheArray[n];
  ...
}
```

What is now the symbolic value of v? In this case, there is no value stored in the vector under the symbolic index n and a sequence of branches for each symbolic index stored previously is produced. Informally

$$\Big(\; n == k \implies SE(i+1,\ldots,\sigma[v/1]) \; \Big) \wedge \Big(\; n == l \implies SE(i+1,\ldots,\sigma[v/2]) \; \Big)$$

where $SE(i+1,\ldots,\sigma[v/1])$ means the recursive call symbolic evaluator for the next instruction where the symbolic value of v is 1.

The last branch corresponds to the fact that the indexing expression can refer to an index not equal to **any** indexes previously stored in the vector:

$$(n \neq k \;\wedge\; n \neq l) \implies SE(i+1,\ldots,\sigma[v/TheArray[n]])$$

which means that the evaluation continues in a state where v has the symbolic value from the array but with unknown exact value, denoted as $TheArray[n]$.

Note, that symbolic indexing expression can be arbitrarily complicated, for example:

```
int Foo( int[] TheArray, int k, int l )
{
  ...
  TheArray[l] = 2;
  ...
  n = k * k + l;
  int v = TheArray[n];
  ...
}
```

In this case, the indexing expression is $k * k + l$.

The `stelem` instruction follows the same pattern. It stores the value in the symbolic vector if the vector is empty or a value of the same index has been stored before.

If however the indexing expression is not syntactically equal to any symbolic indexes stored in the symbolic vector, a sequence of branching instructions is generated, each one taking the equality of symbolic indexes as the premise. What is interesting is that in such case there are two updates to the array - the value to be stored is saved under the actual index and the value of the existing index is removed from the array.

And, similariliy to the `ldelem case` it's possible that the symbolic index is not equal to any symbolic indexes previously stored in the symbolic vector. In such case the value is stored in a new slot in the symbolic vector but the premise has to explicitely capture the fact that the index is not equal to indexes of previously stored values (please refer to the example on Page 114).

Consider the following example:

```
int Foo( int[] TheArray )
{
  ...
  TheArray[k] = 1; /* 1 */
  TheArray[n] = 2; /* 2 */
  ...
  int v = TheArray[k]; /* 3 */
  ...
}
```

The first value, 1, is stored under the symbolic index k. The second value, 2, generates two branches.

The first branch assumes that $k = n$ so the value is stored under the symbolic index n and the value from the index k is removed from the array. When the value is then read from the array using k as the index, the `ldelem` semantics generates yet another two subbranches. The first subbranch has the compatible premise, $k = n$, and the value 2 will be returned. The second subbranch has a contradictory premise, $k \neq n$, so the remaining part of the predicate will always hold (as a result of two contradictory premises which correspond to impossible control flow sequence).

The second branch assumes that $k \neq n$ and the vector stores two different values under two different indexes.

Virtual Calls

Any time a virtual method is called (`callvirt`) the object's vtable is used to determine the actual method to be called.

```
class Base {
  public virtual void Foo() { };
}

class Derived : Base {
  public override void Foo() { };
}

...
void Bar( Base instance )
{
  instance.Foo();
}
```

In the above example the actual method called inside the `Bar` method cannot be determined statically since it depends on the caller. And since both `Base::Foo` and `Derived::Foo` can have **different** specifications, XMS must address this issue in a special way.

The solution is proposed by the Design By Contract so called **Subcontracting Rule**. Subcontracting means that the signature of overridden method must strictly depend on signature of the virtual one.

To formally define the subcontracting we need some terminology. We will say that the predicate P is **stronger** than the predicate Q if $P \implies Q$.

For example, predicate $P := x > 5$ is stronger than $Q := x > 0$. Note that the strongest predicate is `false` and the weakest is `true`. The intuition is that the more strong a predicate is, the harder to satisfy it becomes.

From the supplier point of view, the stronger precondition, the better - there are simpler cases to handle. For example, it helps the supplier to be sure that the method has been invoked with $Person.age > 0 \ \wedge \ Person.age < 20$ than the sole $Person.age > 0$. The stronger the precondition the easier it is to satisfy the postcondition.

And again, from the supplier point of view, the weaker postcondition, the better - the implementation needs not to handle difficult cases. For example, it is much easier to satisfy $Person.age > 0$ than $Person.age > 0 \ \wedge \ Person.age < 20$.

From the client point of view the situation is completely opposite. The weaker precondition, the easier it is to satisfy it. The stronger the postcondition, the more knowledge about the computation is needed.

The object-oriented methodology defines so called **Liskov Substitution Principle**:

> Any object of the derived class must be usable in place of a base class object

This principle lays the base for a subcontracting rule:

- precondition of derived class must be **weaker** than the precondition of base class
- postcondition of derived class must be **stronger** than the postcondition of base class

In any other case the Substitution Principle would not be satisfied. This situation is handled by the *VCI* part of the Verification Condition (page 54).

Example 5.6 Consider the following example which breaks the Substitution Principle.

```
class Person {

  Pre  := Age > 20
  virtual void Hire() { }
  }

}

class Manager : Person {

  Pre  := Age > 20 && Education == Higher
  override void Hire() { }

}
```

In this case, for any code which accepts the `Person` object, the precondition `Age > 20` must be satisfied to call the `Hire` method. However, the `Manager` class cannot be substituted for `Person` because the precondition of the `Hire` method is stronger and it would not necessarily be satisfied upon the invocation.

5.3.3 The Safety Theorem

The soundness of the verification condition algorithm is stated in a following theorem which is an instantiation of the *meta* theorem 3.1.

Theorem 5.1 (Soundness of XMS Safety Policy)**.** *If the verification condition for a given module M is valid, i.e.* $\models VC(\mathcal{M})$ *then all executions of any module methods are correct with respect to the Design By Contract security policy 5.1, i.e.* $Safe_{SC}(\mathcal{M})$.

The formal proof of this theorem is the only missing piece of XMS framework. Note that although the above formulation is clear and concise, the proof presented in appendix B is rather technical and must precisely consider all the details presented above.

Table 5.1: XMS Symbolic Evaluator for Design By Contract

F_i	Condition / action	Symbolic state	Verification condition	$\mathcal{L}$
Instructions with non-empty invariants				
any, where $Inv_F(i) = (Pred, Var, k)$	$i \notin Dom(\mathcal{L}) \ \wedge \ b =$ `true`	$(i, \ldots, u_1 \ldots u_k \cdot s)$	$\sigma(Inv_F(i)) \wedge$ $\forall v'_0, \ldots, v'_m . u'_1 \ldots u'_k . \sigma'(Inv_F(i)) \Rightarrow$ $SE(i, \sigma', (i, \sigma, \emptyset, 0) \cdot \mathcal{L},$ `false`) **where** : $\sigma' = (\ldots, l_V(v_i \mapsto v'_i{}^{\,v_i \in Var}), u'_1 \ldots u'_k \cdot s)$ $v'_0, \ldots, v'_m, u'_1 \ldots, u'_k$ are fresh variables	
any, where $Inv_F(i) = (Pred, Var, k)$	$i \in Dom(\mathcal{L})$	(i, σ)	$\sigma(Pred)$	$\mathcal{L}(i) \notin Var \Rightarrow$ `fail` $\lvert\mathcal{L}(i)\rvert > \lvert k \rvert \Rightarrow$ `fail`
Control Flow Instructions				
`br l`	$l < i \ \wedge \ Inv_F(i) = \epsilon \Rightarrow$ `fail`		$SE(l, \sigma, \mathcal{L})$	
`brtrue l`	$l < i \ \wedge \ Inv_F(i) = \epsilon \Rightarrow$ `fail`	$(i, \ldots, v \cdot s)$	$\sigma(v) = 0 \Rightarrow SE(i+1, \ldots, s, \mathcal{L}) \wedge$ $\sigma(v) \neq 0 \Rightarrow SE(l, \ldots, s, \mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$
`brfalse l`	$l < i \ \wedge \ Inv_F(i) = \epsilon \Rightarrow$ `fail`	$(i, \ldots, v \cdot s)$	$\sigma(v) \neq 0 \Rightarrow SE(i+1, \ldots, s, \mathcal{L}) \wedge$ $\sigma(v) = 0 \Rightarrow SE(l, \ldots, s, \mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$
`bne l`	$l < i \ \wedge \ Inv_F(i) = \epsilon \Rightarrow$ `fail`	$(i, \ldots, u \cdot v \cdot s)$	$\sigma(u) = \sigma(v) \Rightarrow SE(i+1, \ldots, s, \mathcal{L}) \wedge$ $\sigma(u) \neq \sigma(v) \Rightarrow SE(l, \ldots, s, \mathcal{L})$	$\mathcal{L}(s \leftarrow -2)$
`bgt l`	$l < i \ \wedge \ Inv_F(i) = \epsilon \Rightarrow$ `fail`	$(i, \ldots, u \cdot v \cdot s)$	$\sigma(u) \leq \sigma(v) \Rightarrow SE(i+1, \ldots, s, \mathcal{L}) \wedge$ $\sigma(u) > \sigma(v) \Rightarrow SE(l, \ldots, s, \mathcal{L})$	$\mathcal{L}(s \leftarrow -2)$
`blt l`	$l < i \ \wedge \ Inv_F(i) = \epsilon \Rightarrow$ `fail`	$(i, \ldots, u \cdot v \cdot s)$	$\sigma(u) \geq \sigma(v) \Rightarrow SE(i+1, \ldots, s, \mathcal{L}) \wedge$ $\sigma(u) < \sigma(v) \Rightarrow SE(l, \ldots, s, \mathcal{L})$	$\mathcal{L}(s \leftarrow -2)$
`bge l`	$l < i \ \wedge \ Inv_F(i) = \epsilon \Rightarrow$ `fail`	$(i, \ldots, u \cdot v \cdot s)$	$\sigma(u) < \sigma(v) \Rightarrow SE(i+1, \ldots, s, \mathcal{L}) \wedge$ $\sigma(u) \geq \sigma(v) \Rightarrow SE(l, \ldots, s, \mathcal{L})$	$\mathcal{L}(s \leftarrow -2)$
`ble l`	$l < i \ \wedge \ Inv_F(i) = \epsilon \Rightarrow$ `fail`	$(i, \ldots, u \cdot v \cdot s)$	$\sigma(u) > \sigma(v) \Rightarrow SE(i+1, \ldots, s, \mathcal{L}) \wedge$ $\sigma(u) \leq \sigma(v) \Rightarrow SE(l, \ldots, s, \mathcal{L})$	$\mathcal{L}(s \leftarrow -2)$
`ceq`		$(i, \ldots, u \cdot v \cdot s)$	$\sigma(u) = \sigma(v) \Rightarrow SE(i+1, \ldots, 1 \cdot s, \mathcal{L}) \wedge$	$\mathcal{L}(s \leftarrow -1)$

F_i	Condition / check	Symbolic state	Verification condition	Action
			$\sigma(u) \neq \sigma(v) \Rightarrow SE(i+1,\ldots,0\cdot s,\mathcal{L})$	
`cgt`		$(i,\ldots,u\cdot v\cdot s)$	$\sigma(u) > \sigma(v) \Rightarrow SE(i+1,\ldots,1\cdot s,\mathcal{L})\wedge$ $\sigma(u) \leq \sigma(v) \Rightarrow SE(i+1,\ldots,0\cdot s,\mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$
`clt`		$(i,\ldots,u\cdot v\cdot s)$	$\sigma(u) < \sigma(v) \Rightarrow SE(i+1,\ldots,1\cdot s,\mathcal{L})\wedge$ $\sigma(u) \geq \sigma(v) \Rightarrow SE(i+1,\ldots,0\cdot s,\mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$
`ret`	$Sig_F = C\ F(\ldots)$	$(i,\ldots,u\cdot s)$	$\sigma(Post_F[u/VALUE])$	
`ret`	$Sig_F =$ `void` $F(\ldots)$	$(i,\ldots,s)$	$\sigma(Post_F)$	
Arithmetical instructions				
`ldc.i4` u		$(i,\ldots,s)$	$SE(i+1,\ldots,u\cdot s,\mathcal{L})$	$\mathcal{L}(s \leftarrow 1)$
`ldc.r8` u		$(i,\ldots,s)$	$SE(i+1,\ldots,u\cdot s,\mathcal{L})$	$\mathcal{L}(s \leftarrow 1)$
`dup`		$(i,\ldots,v\cdot s)$	$SE(i+1,\ldots,v\cdot v\cdot s,\mathcal{L})$	$\mathcal{L}(s \leftarrow 1)$
`pop`		$(i,\ldots,v\cdot s)$	$SE(i+1,\ldots,s,\mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$
`add`		$(i,\ldots,u\cdot v\cdot s)$	$SE(i+1,\ldots,(u+v)\cdot s,\mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$
`sub`		$(i,\ldots,u\cdot v\cdot s)$	$SE(i+1,\ldots,(u-v)\cdot s,\mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$
`mul`		$(i,\ldots,u\cdot v\cdot s)$	$SE(i+1,\ldots,(u*v)\cdot s,\mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$
`div`		$(i,\ldots,u\cdot v\cdot s)$	$SE(i+1,\ldots,(u/v)\cdot s,\mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$
`rem`		$(i,\ldots,u\cdot v\cdot s)$	$SE(i+1,\ldots,(u\%v)\cdot s,\mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$
`and`		$(i,\ldots,u\cdot v\cdot s)$	$SE(i+1,\ldots,(u\wedge v)\cdot s,\mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$
`add`		$(i,\ldots,u\cdot v\cdot s)$	$SE(i+1,\ldots,(u\vee v)\cdot s,\mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$
`xor`		$(i,\ldots,u\cdot v\cdot s)$	$SE(i+1,\ldots,(u\oplus v)\cdot s,\mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$
`neg`		$(i,\ldots,v\cdot s)$	$SE(i+1,\ldots,-v\cdot s,\mathcal{L})$	
`not`		$(i,\ldots,v\cdot s)$	$SE(i+1,\ldots,\neg v\cdot s,\mathcal{L})$	
Addressing Fields, Arguments and Local Variables				
`ldarg` v		$(i,\ldots,s)$	$SE(i+1,\ldots,l_A(v)\cdot s,\mathcal{L})$	$\mathcal{L}(s \leftarrow 1)$
`starg` v		$(i,l_A,\ldots,u\cdot s)$	$SE(i+1,l_A[v\mapsto u],\ldots,s,\mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$ $\mathcal{L}(V \leftarrow \{v\})$
`ldloc` n		$(i,\ldots,s)$	$SE(i+1,\ldots,l_V(n)\cdot s,\mathcal{L})$	$\mathcal{L}(s \leftarrow 1)$
`stloc` n		$(i,\ldots,l_V,u\cdot s)$	$SE(i+1,\ldots,l_V[n\mapsto u],s,\mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$ $\mathcal{L}(V \leftarrow \{v_n\})$

F_i	Condition / check	Symbolic state	Verification condition	Action
`ldfld` $T :: f$		$(i, \ldots, p \cdot s)$	$SE(i+1, \ldots, p(f) \cdot s, \mathcal{L})$	$\mathcal{L}(s \leftarrow 1)$
`stfld` $T :: f$		$(i, \ldots, u \cdot p \cdot s)$	$SE(i+1, p[f \mapsto u], \ldots, s, \mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$ $\mathcal{L}(V \leftarrow \{p :: f\})$
`ldsfld` $T :: f$		$(i, \ldots, s)$	$SE(i+1, \ldots, T(f) \cdot s, \mathcal{L})$	$\mathcal{L}(s \leftarrow 1)$
`stsfld` $T :: f$		$(i, \ldots, u \cdot s)$	$SE(i+1, T[f \mapsto u], \ldots, s, \mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$ $\mathcal{L}(V \leftarrow \{T :: f\})$
Calling methods				
`call` G `callvirt` G	$Sig_G = C\ G(a_0, \ldots, a_n)$ G is static	$(i, \ldots, u_n \ldots u_0 \cdot s)$	$\sigma^G(Pre_G) \wedge$ $\forall u.\sigma^G(Post_G[u/VALUE]) \Rightarrow$ $SE(i+1, \sigma', \mathcal{L})$ **where** : $\sigma^G = (l_A^G(a_i \mapsto u_i), l_V^G(v_i \mapsto 0), \epsilon)$ u is a fresh variable $\sigma' = (\ldots, u \cdot s)$	$\mathcal{L}(s \leftarrow -(n+1))$
`call` G `callvirt` G	$Sig_G =$ `void` $G(a_0, \ldots, a_n)$ G is static	$(i, \ldots, u_n \ldots u_0 \cdot s)$	$\sigma^G(Pre_G) \wedge$ $\forall u.\sigma^G(Post_G) \Rightarrow$ $SE(i+1, \sigma, \mathcal{L})$ **where** : $\sigma^G = (l_A^G(a_i \mapsto u_i), l_V^G(v_i \mapsto 0), \epsilon)$	$\mathcal{L}(s \leftarrow -(n+2))$
`call` G `callvirt` G	$Sig_G = C\ G(a_0, \ldots, a_n)$ G is instance	$(i, \ldots, u_n \ldots u_0 \cdot p \cdot s)$	$\sigma^G(Pre_G) \wedge$ $\forall u.\sigma^G(Post_G[u/VALUE, p/THIS]) \Rightarrow$ $SE(i+1, \sigma', \mathcal{L})$ **where** : $\sigma^G = (l_A^G(p, a_i \mapsto u_i), l_V^G(v_i \mapsto 0), \epsilon)$ u is a fresh variable $\sigma' = (\ldots, u \cdot s)$	$\mathcal{L}(s \leftarrow -(n+2))$
`call` G `callvirt` G	$Sig_G =$ `void` $G(a_0, \ldots, a_n)$ G is instance	$(i, \ldots, u_n \ldots u_0 \cdot p \cdot s)$	$\sigma^G(Pre_G) \wedge$ $\forall u.\sigma^G(Post_G[p/THIS]) \Rightarrow$ $SE(i+1, \sigma, \mathcal{L})$ **where** :	$\mathcal{L}(s \leftarrow -(n+3))$

F_i	Condition / check	Symbolic state	Verification condition	Action
			$\sigma^G = (l_A^G(p, a_i \mapsto u_i), l_V^G(v_i \mapsto 0), \epsilon)$	
Classes and value types				
`ldnull`		$(i, \ldots, s)$	$SE(i+1, \ldots, \texttt{null} \cdot s, \mathcal{L})$	$\mathcal{L}(s \leftarrow 1)$
`ldstr` *str*		$(i, \ldots, s)$	$SE(i+1, \ldots, str \cdot s, \mathcal{L})$	$\mathcal{L}(s \leftarrow 1)$
`newobj` C	$Sig_C = .\texttt{ctor}(u_0, \ldots, u_n)$ Pre_C = `.ctor` precond. $Post_C$ = `.ctor` postcond.	$(i, \ldots, u_n \ldots u_0 \cdot s)$	$\sigma^C(Pre_C) \wedge$ $\sigma^C(Post_C[p/THIS]) \Rightarrow$ $SE(i+1, \ldots, p \cdot \sigma', \mathcal{L})$ **where** : $\sigma^C = (l_A^C(a_i \mapsto u_i), l_V^C(v_i \mapsto 0), \epsilon)$ p is a fresh symbolic object	$\mathcal{L}(s \leftarrow -(n+1))$
Vectors				
`newarr` T		$(i, \ldots, k \cdot s)$	$SE(i+1, \ldots, a_T[Length \mapsto k] \cdot s, \mathcal{L})$ **where** : a is a fresh symbolic vector of type T	$\mathcal{L}(s \leftarrow 0)$
`ldlen`		$(i, \ldots, a \cdot s)$	$SE(i+1, \ldots, a(Length) \cdot s, \mathcal{L})$	$\mathcal{L}(s \leftarrow 0)$
`ldelem`	$Dom(a) = \emptyset$	$(i, \ldots, e \cdot a \cdot s)$	$SE(i+1, \ldots, a[e] \cdot s, \mathcal{L})$ **where** : $a[e]$ denotes an unknown element of the vector	$\mathcal{L}(s \leftarrow -1)$
`ldelem`	$Dom(a) \neq \emptyset \wedge e \in Dom(a)$	$(i, \ldots, e \cdot a \cdot s)$	$SE(i+1, \ldots, a(e) \cdot s, \mathcal{L})$	$\mathcal{L}(s \leftarrow -1)$
`ldelem`	$Dom(a) \neq \emptyset \wedge e \notin Dom(a)$	$(i, \ldots, e \cdot a \cdot s)$	$e = u_0 \implies SE(i+1, \ldots, a(u_0) \cdot s, \mathcal{L}) \wedge$ $e = u_1 \implies SE(i+1, \ldots, a(u_1) \cdot s, \mathcal{L}) \wedge$ $\ldots$ $e = u_n \implies SE(i+1, \ldots, a(u_n) \cdot s, \mathcal{L}) \wedge$ $(e \neq u_1 \wedge \ldots \wedge e \neq u_n) \implies$ $SE(i+1, \ldots, a(e) \cdot s, \mathcal{L})$ **where** : $Dom(a) = (u_0, \ldots, u_n)$	$\mathcal{L}(s \leftarrow -1)$
`stelem`	$Dom(a) = \emptyset \vee e \in Dom(a)$	$(i, \ldots, e \cdot u \cdot a \cdot s)$	$SE(i+1, \ldots a[u \mapsto e] \ldots s, \mathcal{L})$	$\mathcal{L}(s \leftarrow -3)$
`stelem`	$Dom(a) \neq \emptyset \wedge e \notin Dom(a)$	$(i, \ldots, e \cdot u \cdot a \cdot s)$	$e = u_0 \implies$	

F_i	Condition / check	Symbolic state	Verification condition	Action
			$SE(i+1,\ldots,a[u \mapsto e, u_0 \mapsto \epsilon] \cdot s, \mathcal{L}) \wedge$ $e = u_1 \implies$ $SE(i+1,\ldots,a[u \mapsto e, u_1 \mapsto \epsilon] \cdot s, \mathcal{L}) \wedge$ $\ldots$ $e = u_n \implies$ $SE(i+1,\ldots,a[u \mapsto e, u_n \mapsto \epsilon] \cdot s, \mathcal{L}) \wedge$ $(e \neq u_1 \wedge \ldots \wedge e \neq u_n) \implies$ $SE(i+1,\ldots,a[u \mapsto e] \cdot s, \mathcal{L})$ **where** : $Dom(a) = (u_0,\ldots,u_n)$	$\mathcal{L}(s \leftarrow -1)$

Chapter 6

Towards High-Level Languages

6.1 From MSIL to High-Level Languages

The .NET paradigm unifies many programming languages at MSIL level. Whether you use C#, VB.NET, Managed C++ or any other .NET language, your code can closely cooperate with any other .NET code.

Since the Verification Condition Generator works at MSIL level, it cannot determine which language was used to produce MSIL binary. And no matter if a binary was produced by IL compiler, C# compiler or any other language compiler, it should be certifiable in the uniform way.

The goal of ”lifting” the certification framework from MSIL to a high-level language is then executed under two assumptions:

- A high-level language developer should not be forced to learn MSIL language. In particular, a solution where a high-level code is first compiled to MSIL and then manually certified is unacceptable. Certificates should be then easily applicable to a high-level language code.

- A high-level compiler should not require any major changes to support the certification. In fact, it would be perfect, if the high-level compiler did not require **any** changes. In particular, existing high-level language compilers should not damage certificates that were applied to high-level code.

At first look these assumptions do not seem troublesome - specifications are provided in attributes and attributes should not be damaged during the translation from high-level language to MSIL.

Unfortunately, it seems that comparing to other security policies, Design By Contract is non-trivial to be applied to high-level languages. There are two important difficulties that have to be addressed:

- XMS invariants have the form $Inv_F(i) = (P, \ldots)$ where i is the MSIL instruction number and P is the invariant predicate. It could be however extremely difficult to determine the MSIL instruction number for given high-level instruction, since it would require a deep knowledge of compiler transformation routines.

- During the compilation to MSIL, names of local variables are omitted. Whenever such local variable plays any role in one of the predicates (in a loop invariant for example) it cannot be referred with its name since the name is lost during the compilation.

The first difficulty can be addressed with a clever technical trick. We would like to avoid attributing invariant predicates with MSIL instruction numbers. We would rather like to have an ordered set of invariants:

$$Invs_F = (P_0, \ldots, P_n)$$

and somehow infer Inv_F from it by mapping consecutive invariants to instructions that need invariants.

This goal can be achieved with additional scan of the binary code which could discover instructions $I = (i_0, \ldots, i_k)$ that are targets for backward jumps.

We could then take:

$$Inv_F(i) = \begin{cases} P_j & \text{if } i = i_j \text{ for some } j \text{ and } j \leq n \\ \epsilon & \text{in other case} \end{cases}$$

Note that although this trick seems to be restrictive at first look (these and **only** these instructions which are targets for backward jumps will have invariants mapped) in practice it works perfectly since invariants are used by the Symbolic Evaluator **only** for such kind of instructions.

The second difficulty can be addressed by ”virtually” renaming consecutive local variables to any set of fixed names ($v_0, \ldots, v_n$ in case of XMS) and using these ”virtual” names in specifications by a high-level language developer.

Example 6.1 Consider following C# method used for calculating the sum of values from 0 to n.

```
  [XMS_Spec(                    /* XMS attribute in metadata, including */
  "n >= 0",                     /* precondition, */
  "VALUE=sum(0, n)",            /* postcondition, */
  "V_0=sum(0, V_1) & n >= V_1"] /* and invariant */
public int Foo( int n )
{
  int retval = 0;
  for ( int k=0; k<=n; k++ )
  {
    retval += k;
  }

  return retval;
  }
```

Note that there is one loop in above code which requires an invariant and there are two local variables, *retval* and *k*, which have to be ”virtually” renamed to v_0 and v_1.

Note also that both the postcondition and the invariant use *sum* (Σ) function which **must** be defined in the logic and handled by the theorem prooving layer of the infrastructure. In the above example $sum(0, n)$ denotes $\Sigma_{i=0}^{n} i$.

Fortunately, the high-level language compiler translates consecutive high-level local variables to consecutive MSIL local variables. In above example, the *retval* variable is defined as first local variable and it will become the first local variable in MSIL output and so on.

This is why the `V_0` and `V_1` names are used in the invariant specified in the method's attribute. The `V_0` refers to *retval* and `V_1` refers to *k*.

The Verification Condition for this method is as follows[1]:

```
forall n. (n >= 0 => 0=sum(0, 0) & n >= 0 &
  forall V_0_. forall V_1_.
  V_0_=sum(0, V_1_) &
  n >= V_1_=>
    ((V_1_<n => (V_0_+(V_1_+1))=
      sum(0, (V_1_+1)) & n >= (V_1_+1)) &
     (V_1_>=n => V_0_=V_0_ &
       V_0_=sum(0, n))))
```

Note that the invariant was first checked using the actual state of the symbolic evaluation [the $0 = sum(0,0) \wedge n >= 0$ part] and then variables which are modified inside the loop body were replaced with new, fresh, universally quantified values ($V_{0'}$ and $V_{1'}$) and the invariant was first used as a premise [$\forall V_{0'}.\forall V_{1'}.V_{0'} = sum(0, V_{1'}) \wedge n >= V_{1'} => \ldots$] and then verified after the loop iteration [$(V_{0'} + (V_{1'} + 1)) = sum(0, (V_{1'} + 1)) \wedge n >= (V_{1'} + 1)$].

6.2 Common Certificate Specification

Both above technical tricks require that the high-level language satisfies two important conditions. These conditions are **essential** for the "lifting" process to work, so we will formulate them as the **Common Certificate Specification** (by analogy to Common Language Specification and Common Type System, two fundamental .NET paradigms). The Common Certificate Specification is as follows:

Variable Ordering Consecutive high-level language local variables become consecutive MSIL local variables.

Structure of Loops High level language loops become MSIL loops with corresponding structure.

While the above specification does not look formal enough, we are not going to make it formal. It is because some important existing compilers (like the C# compiler) fulfill both these requirements and the CCS formulation should be treated as a set of guidelines for new compilers.

Both requirements are crucial for proper translation of loop invariants between a high-level language and MSIL. In the example above the loop invariant refers to variables `sum` and `k` but in MSIL they become V_0 and V_1. Since there is only one loop in C# code, only one loop invariant should be supplied. VCGen will automatically detect the instruction which correspond to the

[1] Actual predicate can differ from this one since compiler translation routines depend on compiler version and compiling options

invariant. Note that the actual number of instructions could depend on the optimization level used by the C# compiler.

In fact, the main reason that makes the ”lifting” possible is that .NET high-level language compilers follow few simple and obvious patterns while producing MSIL from high-level code. This is not a coincidence and chances are that future compilers will also behave in similar way because MSIL is not a platform-native language – it is the Just-In-Time compiler which does most of fancy optimizations while translating the MSIL to platform-native language.

Of course this ”simple translation” rule applies mainly to major enterprise languages for the .NET platform - C# and VB.NET. Other languages with different translation schemes must find their own way to integrate with XMS. There are three possible **integration strategies**:

no integration or limited integration Developers are forced to consult the compiler output to find exact MSIL structure and then put appropriate attributes either at language level or at MSIL level.

attribute integration The language recognizes XMS attributes and knowing its own translation schemes puts the attributes in appropriate places inside MSIL.

language integration The language syntax is augmented with first-class contract expressions which are compiled either as direct expressions used during the dynamic testing or as static XMS attributes used during static verification.

6.3 High-Level Compiler Translation Schemes

In this section we would like to present few specific C# translation schemes to informally argue that this is possible to adopt XMS at the level of C# with no major difficulties.

6.3.1 Variable Ordering

Local variables declared in a C# method’s body usually become consecutive variables in MSIL body. Look at following example:

C#	MSIL

`public static void F()`
`{`
`  int i, j, k;`
 	
`  i=0;`
 	
`  j=1;`
 	
`  k=2;`
 	
`}`
	`.method public static void F()`
	`{`
	`  .locals init`
	`  (int32 V_0, int32 V_1, int32 V_2)`
	`  ldc.i4.0`
	`  stloc.0`
	`  ldc.i4.1`
	`  stloc.1`
	`  ldc.i4.2`
	`  stloc.2`
	
	`  ret`
	`}`

There are three local variables in the C# code which become three consecutive variables in the MSIL code and writing a predicate referring to any of these variables at the C# level would bring no difficulties.

Things became slightly more complicated when the .NET 2.0 has been released. It seems that the internal compiler routines works differently when optimization is enabled/disabled. In case of disabled optimizations the code is bloated with auxiliary variables which store results of logical expressions or the result value of the method.

Consider following C# method:

```
public int Abs( int x )
{
  if ( x > 0 )
    return x;
  else
    return -x;
}
```

which would have the precondition `true` and obvious postcondition $VALUE \geq 0$

When optimizations are enabled it produces following MSIL code:

```
.method public hidebysig
 instance int32 Abs(int32 x) cil managed
{
    .maxstack 8
    L_0000: ldarg.1       // x onto the stack
    L_0001: ldc.i4.0      // 0 onto the stack
    L_0002: ble.s L_0006  // x <=0 ? jump to 0006
    L_0004: ldarg.1       // x onto the stack
    L_0005: ret           // return x
    L_0006: ldarg.1       // x onto the stack
    L_0007: neg           // -x on the stack
    L_0008: ret           // return -x
}
```

which gives following Verification Condition:

$$\forall x.\texttt{true} \implies (((x \geq 0) => (x \geq 0)) \wedge ((x < 0) \implies (-x \geq 0)))$$

Note that in the above example it is the `ldarg.1` which loads the first formal method parameter onto the stack and not the `ldarg.0`.

It is because the `ldarg.0` would load the `this` reference onto the stack since it is always passed as first, hidden parameter to instance methods.

With optimizations disabled the C# code is translated as:

```
.method public hidebysig
 instance int32 Abs(int32 x) cil managed
{
    .maxstack 2
    .locals init (
```

```
        [0] int32 num,
        [1] bool flag)
    L_0000: nop               // do nothing
    L_0001: ldarg.1           // x onto the stack
    L_0002: ldc.i4.0          // 0 onto the stack
    L_0003: cgt               // x > 0  on the stack
    L_0005: ldc.i4.0          // 0 onto the stack
    L_0006: ceq               // x <= 0 on the stack
    L_0008: stloc.1           // store the bool expression
    L_0009: ldloc.1
    L_000a: brtrue.s L_0010   // x <= 0 ? jump 0010
    L_000c: ldarg.1           // x onto the stack
    L_000d: stloc.0           // store it in local variable
    L_000e: br.s L_0015       // jump 0015
    L_0010: ldarg.1           // x onto the stack
    L_0011: neg               // -x onto the stack
    L_0012: stloc.0           // store it in local variable
    L_0013: br.s L_0015       // jump 0015
    L_0015: ldloc.0           // local variable onto the stack
    L_0016: ret               // return
}
```

which gives different Verification Condition:

$$
\begin{aligned}
&\forall x.\mathtt{true} \implies \\
&(((x < 0) \implies \\
&\qquad (((\mathtt{true} = 0) \implies (x \geq 0)) \wedge \\
&\qquad ((\mathtt{true} \neq 0) \implies (-x \geq 0)))) \wedge \\
&((x \geq 0) \implies \\
&\qquad (((\mathtt{false} = 0) \implies (x \geq 0)) \wedge \\
&\qquad ((\mathtt{false} \neq 0) \implies (-x \geq 0)))))
\end{aligned}
$$

The first observation is that two different translation schemes lead to different MSIL code and two different Verification Conditions. However, both should hold or not in the same time since they correspond to the same high-level source code. This is probably one of the most interesting properties of the XMS static verification.

In our case, both predicates are true and they exactly reflect the structure of two different translation schemes - note how the bloated structure of the unoptimized code produces the hierarchy of assumptions which lead to branches which always hold because of false assumptions.

The second observation is that these two translation schemes are based on different rules. There are two `ret` instructions in the optimized version while the unoptimized scheme accumulates the returning value in an auxiliary variable. The unoptimized scheme uses also completely different pattern for the conditional expression with an auxiliary variable to store the result of the logical expression. As a result - there are two auxiliary variables in the unoptimized scheme and no auxiliary variables in the optimized one.

Concluding this example - while the high-level language developer must be aware of such issues and while it looks cumbersome at first sight, it should bring no major problems when using the XMS at a high-level language level, at least when the language is compatible with the Common Certificate Specification set of obligations.

6.3.2 Assignments, Expressions

Expressions in the high-level language code expand to sequences of MSIL instructions. Assignments translate to one of `stloc`, `starg`, `stelem`, `stfld` or `stsfld` instructions. For example:

```
public int Foo( int a, int b, int c )
{
  return b * b - 4 * a * c;
}
```

with following postcondition: $VALUE = b * b - 4 * a * c$, produces following MSIL sequence:

```
.method public hidebysig
instance int32 Foo(int32 a, int32 b, int32 c) cil managed
{
    .maxstack 8
    L_0000: ldarg.2  // b onto the stack
    L_0001: ldarg.2
    L_0002: mul      // b*b on the stack
    L_0003: ldc.i4.4 // 4 onto the stack
    L_0004: ldarg.1  // a onto the stack
    L_0005: mul      // 4*a on the stack
    L_0006: ldarg.3  // c onto the stack
    L_0007: mul      // 4*a*c onto the stack
    L_0008: sub      // b*b-4*a*c on the stack
    L_0009: ret
}
```

and following Verification Condition:

$$\forall a.\forall b.\forall c.true => (((b * b) - ((4 * a) * c)) = ((b * b) - ((4 * a) * c)))$$

6.3.3 Loops

The C# language has inherited three loop constructions from its ancestor, the C language:

- ```
  while ( bool_expression )
  {
    ..loop body..
  };
  ```
- ```
  do
  {
    ..loop body..
  } while ( bool_expression );
  ```
- ```
  for ( init_expression; bool_expression; loop_expression )
  {
    ..loop body..
  }
  ```
  ```

C#	MSIL

int i;
...
for (i=0; i<1; i++)
{
...body...
}
	.locals init (int32 V_0)
	...
	ldc.i4.0
	stloc.0 // i=0
	br loop_end
	
	loop_body:
	...body...
	
	ldloc.0
	ldc.i4.1
	add
	stloc.0 // i=i+1
	
	loop_end:
	ldloc.0
	ldc.i4.1
	blt loop_body // jump if i<1

Figure 6.1: Translation scheme for `for` instruction

Example translation scheme for the `for` loop is presented in Figure 6.1. At first the loop initialization takes place, then execution code jumps to the loop condition checking and depending on the test the loop is executed or ends. The similarity between the C# and MSIL loop structures makes it possible to formulate loop invariants at C# level which still makes sense after the code is translated to MSIL by the C# compiler.

6.4 Other High-Level language features

6.4.1 Class Invariants

As we have already mentioned the class invariant is a predicate which obligates the supplier to make it satisfiable whenever the client has an access to an object instance. It is a common belief that class invariants are redundant since they can be encoded as parts of pre- and postconditions.

Example 6.2 Consider following C# class:

```
[ClassInvariant( "this.deposit >= 0" )]
class Account {

  int deposit;

  [Precondition( "Money >= 0" )]
  [Postcondition( "true" )]
  void Add( int Money ) {
```

```
    deposit += Money;
  }

  [Precondition( "Money >= 0" )]
  [Postcondition( "true" )]
  void Withdraw( int Money ) {
    if ( Money <= deposit )
      deposit -= Money;
    else
      deposit = 0;
  }
}
```

According to this common belief actual specifications could be augmented so that instead

$$Pre_{\texttt{Add}} = \text{Money} \geq 0$$
$$Post_{\texttt{Add}} = \texttt{true}$$

we could have

$$Pre_{\texttt{Add}} = \text{Money} \geq 0 \wedge \text{this.deposit} \geq 0$$
$$Post_{\texttt{Add}} = \text{this.deposit} \geq 0$$

It has been however pointed out that there are obvious cases where such ”shifting” of a class invariant to pre- and postconditions of all instance methods is just incorrect - perhaps the most obvious counter example considers an instance method which performs some initial activities and then calls another instance method. Since the execution of the first method is not completed the instance can be in a state where the class invariant does not hold (the class invariants is obligated to hold when client has access to the instance, not necessarily in the middle of an execution sequence!). Because of that, the augmented precondition of the callee also does not hold and the invocation fails.

XMS handles class invariants at two levels.

Class Invariants in Verification Conditions Since the Verification Condition corresponds to the execution from the client’s point of view, it **can** be augmented with the Class Invariant. The augmented Verification Condition would be:

$$VC(F) = VCI(F) \wedge VCE(F)$$

where:

$$\begin{aligned} VCE(F) &= \forall a_0, \ldots, a_n.\ ClassInv_F \wedge \sigma_0^F \models Pre_F \Rightarrow \\ &\quad (SE(0, \sigma_0^F, \emptyset, \texttt{true}) \wedge \sigma_{ret}(ClassInv_F) \\ VCI(F) &= \text{same as before (section 5.3.1, page 54)} \\ \sigma_0^F &= \text{same as before} \\ \sigma_{ret} &\quad \text{Symbolic Evaluator state at the } \texttt{ret} \text{ instruction} \end{aligned}$$

(reader can refer to Figure 5.1 to compare the original and the augmented version)

Class Invariants in case of method calls Anytime the `call` or `calli` is evaluated by the Symbolic Evaluator a part of the Verification Condition is produced which takes the pre- and postcondition of the callee into consideration. Whenever then a method from **the same** class on **the same** object instance is called the pre- and postconditions are not augmented with the class invariant and when a method from **another** class or the same class but on **another** object instance is called - its class invariant should hold and it is concatenated to both the precondition and the postcondition.

6.4.2 Properties, Indexers

Properties and indexers are not MSIL language constructs. They are considered the *syntax-sugar* and are translated to one or two methods depending on the number of accessors provided.

Example 6.3 Consider following C# code:

```
class Foo {

  int theProperty;

  int TheProperty
  {
    get
    {
      return theProperty;
    }
    set
    {
      theProperty = value;
    }
  }
}
```

The C# compiler produces two methods at MSIL level:

```
.method private hidebysig specialname
 instance int32 get_TheProperty() cil managed
{
    .maxstack 8
    L_0000: ldarg.0
    L_0001: ldfld int32 Foo::theProperty
    L_0006: ret
}

.method private hidebysig specialname
 instance void set_TheProperty(int32 'value') cil managed
{
    .maxstack 8
    L_0000: ldarg.0
    L_0001: ldarg.1
```

```
    L_0002: stfld int32 Foo::theProperty
    L_0007: ret
}
```

Since separate attributes are allowed on both property accessors, specifications for both accessors can be provided for XMS. The only important issue is the compiler generated name for the parameter of the `set` accessor.

The similar pattern applies to indexers. In this case also one or two methods are generated and their parameters depend of actual parameters of the indexer.

6.4.3 Delegates

Delegates are special kind of reference types designed to represent function pointers. They inherit from `System.MulticastDelegate` class, cannot be inherited and must provide two or four mandatory methods. In the former case these two methods are the two-parameter constructor and the `Invoke` method.

The main difference between delegates and function pointers is that delegates are type-safe both at compile and run-time and produce verifiable code where function pointers are considered type-unsafe (for example, converting function pointers with incompatible signature is possible in C with direct cast, however the actual call causes a serious run-time problem which usually terminates the application because of the stack corruption. This mechanism can be also exploited to take the control of the running application).

All member methods of a delegate class are implemented by the runtime so even at the MSIL level their implementation must not be provided. The C# language introduces delegates as first-class language constructs with simplified syntax for the `Invoke` method.

Example 6.4 Following C# code:

```
public class FooClass {
  public delegate int FooDelegate( int x );

  public int FooExample( int n ) {
    return n;
  }

  public int InvokeDelegate() {
    FooDelegate foo = new FooDelegate( FooExample );
    return foo( 1 );
  }
}
```

produces the inner delegate class with four runtime-implemented methods, the constructor and `Invoke` and two complementary `BeginInvoke` and `EndInvoke` methods for asynchronous calls.

```
.class auto ansi sealed nested public FooDelegate
    extends [mscorlib]System.MulticastDelegate
{
```

```
  .method public hidebysig specialname rtspecialname
  instance void .ctor(object 'object', native int 'method')
  runtime managed
  { }

  .method public hidebysig newslot virtual
  instance class [mscorlib]System.IAsyncResult
  BeginInvoke( int32 x,
    class [mscorlib]System.AsyncCallback callback,
    object 'object') runtime managed
  { }

  .method public hidebysig newslot virtual
  instance int32
  EndInvoke(
    class [mscorlib]System.IAsyncResult result)
    runtime managed
  { }

  .method public hidebysig newslot virtual
  instance int32 Invoke(int32 x) runtime managed
  { }
}
```

There are two possibilities a delegate reference can be obtained - it can be created inside a method or obtained from outside (passed as an argument, retrieved from instance field of another object or a static field of some class). Let's examine both cases.

explicit delegate instantiation A delegate can be created in an explicit way using the automatically generated constructor.

```
public int FooExample( int n ) {
  return n;
}

public void ExplicitDelegate() {
  FooDelegate foo = new FooDelegate( FooExample );
  int res = foo( 1 );
}
```

In this case following MSIL is produced for the `ExplicitDelegate` method:

```
.method public hidebysig
 instance void ExplicitDelegate() cil managed
{
  .maxstack 3
  .locals init (
      [0] class DelegateTest/FooDelegate foo)
```

```
  L_0000: ldarg.0              // instance onto the stack
  L_0001: ldftn instance int32 // instance function pointer
    DelegateTest::FooExample(int32) // onto the stack
  L_0007: newobj instance void      // create delegate
    DelegateTest/FooDelegate::.ctor(object, native int)
  L_000c: stloc.0
  L_000d: ldloc.0
  L_000e: ldc.i4.1                  // 1 onto the stack
  L_000f: callvirt instance int32   // invoke the delegate
    DelegateTest/FooDelegate::Invoke(int32)
  L_0014: pop
  L_0015: ret
}
```

Such case is handled by XMS by first storing the function pointer in a **symbolic delegate** reference and then reducing the call to the `Invoke` method to the typical `call` case using the specification from the stored symbolic delegate as a specification of the callee method.

implicit delegate reference The delegate can not only be created in an explicit way as shown in previous case but also the reference can be obtained from elsewhere. For example:

```
public int FooExample( int n ) {
  return n;
}

public void DelegateReference(
  FooDelegate F, int k ) {

  int res = F( k );
}
```

which translates as

```
.method public hidebysig instance void DelegateReference(
  class DelegateTest/FooDelegate F, int32 k) cil managed
{
  .maxstack 8
  L_0000: ldarg.1
  L_0001: ldarg.2
  L_0002: callvirt instance int32
    DelegateTest/FooDelegate::Invoke(int32)
  L_0007: pop
  L_0008: ret
}
```

The easiest trick in this case would be to restrict the possible invocations by forcing the pre- and postcondition of the delegate to fulfill some requirements similar as in subcontracting. Unfortunately, delegates often come from different classes and provide completely different functionality so their specifications are not related in any way.

Such case is then handled by XMS by reducing the call to the `Invoke` method to the typical `call` case but using unknown predicates *Pre*(*k*) and *Post*(*k*) to denote the unknown precondition and the postcondition of the delegate *F* (*k* is supplied as the parameter to the delegate and the *Pre* and *Post* predicates must be parametrized by *k*).

The Verification Condition for the method `DelegateTest` would be then:

$$Pre_{\texttt{DelegateTest}} \implies \ldots Pre(k) \wedge \forall u.Post(k)[u/VALUE] \implies \ldots Post_{\texttt{DelegateTest}}$$

Note that above predicate is a second-order predicate with two functions *Pre* and *Post*. Whether or not such second-order predicate is still provable depends on actual case.

Although the predicate can be unprovable, anytime the `DelegateReference` method is invoked from a place where the *Pre* and *Post* predicates are known from the caller context, an ”instantiation” of the Verification Condition for `DelegateReference` with *Pre* and *Post* replaced by actual predicates could be appended to the global Verification Condition for the module. This would still leave the `DelegateReference` unsafe but could validate any potential calls to it.

Nevertheless, if the reference to the delegate is obtained for example from a static field of another class, the actual function pointer and its specification cannot be determined statically and the code would be probably rejected as unsafe (because no proof of a second-order formula could be found).

Chapter 7

Practical Issues

7.1 The Implementation

A practical implementation of PCC-oriented certification framework requires three key components: the VCGen which builds Verification Conditions for given code modules, an external theorem prover for code producer to build formal proofs of Verification Conditions and an external proof checker for code consumer to verify proofs.

The VCGen was exclusively developed for XMS and runs on the .NET platform itself. It reads .NET binaries, scans method bodies and builds Verification Conditions. Current implementation supports a broad range of MSIL instructions, i.a. arithmetical and control flow instructions, instructions for addressing fields and arguments and instructions for calling methods.

7.1.1 Code-Producer Components

The architecture of the Code-Producer framework is presented in the Figure 7.1.

The framework consists of three major components.

Framework Base consists of the Common Definitions component and Specification Language Parser.

- **Common Definitions** contains mainly the definition of specification attributes. Any XMS .NET module uses these attributes to provide specification for method and class invariants.
- **Specification Language Parser** is the component responsible for providing parsing routines for the specification language. The parser is used by both the static and dynamic verification engines. The parser component contains also the definitions of symbolic objects, symbolic vectors and symbolic indirect expressions used during by the static engine during the symbolic evaluation. All parser-defined classes inherit from the `SymbolicExpression` class which provide the common maintenance such as evaluating the closure, substitution or cloning for symbolic expressions.

  Language parser is written in C# and uses external parser generator.

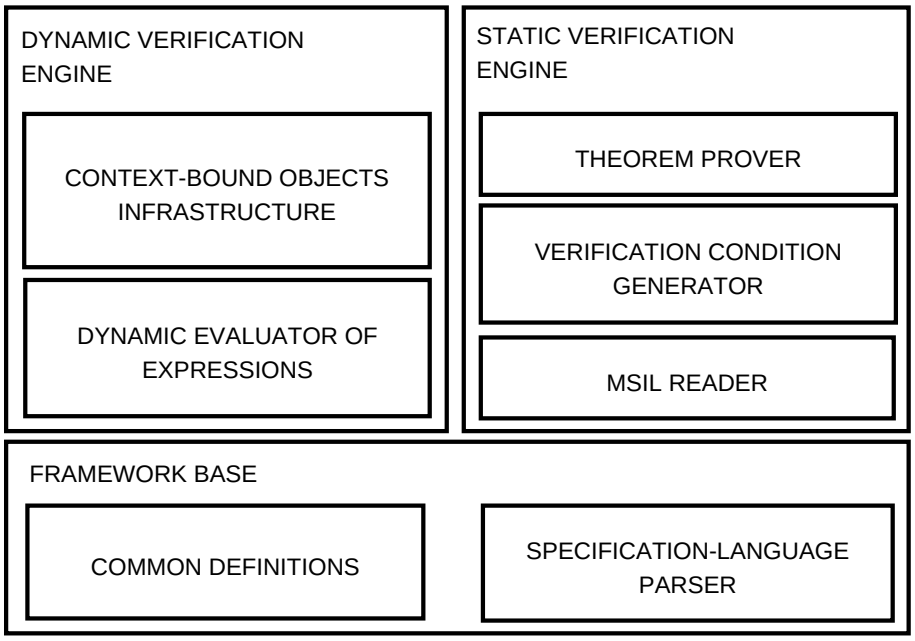

Figure 7.1: XMS Framework at the Code-Producer side

Dynamic Verification Engine consists of the Context-Bound Objects infrastructure and the Dynamic Evaluator of Expressions.

- **Context-Bound Objects** infrastructure contains complete toolbox for .NET code instrumentation. The toolbox is described with details in the Section 5.2.
- Dynamic Evaluator of Expression is responsible for evaluating predicates during the pre- and postprocess phase of the execution under the dynamic verification engine. It uses the .NET dynamic code generation techniques to accept or reject specification predicates by evaluating them. The result of the evaluation is always a boolean value or an exception is thrown when the evaluation cannot be performed.

  Both the Context-Bound Objects infrastructure and the Dynamic Evaluator are written in C# and the code size is about 20kB.

Static Verification Engine consists of the **MSIL Reader**, **Verification Condition Generator** and an external **Theorem Prover**.

- **MSIL Reader** is written in C# and uses both the reflection and a clever trick involving the `GetILAsByteArray` of the `MethodBody` class - the result of the call is not directly usable because its result is a byte array of MSIL data (including opcodes and their arguments), however it has been shown in [42] how the existing opcode enumeration (`System.Reflection.Emit.OpCode`) can be used for extracting actual opcodes and values of their arguments from the MSIL stream.

  The reader is written in C# and the code size is about 20kB.
- **Verification Condition Generator** is responsible for the symbolic evaluation of MSIL methods and producing Verification Conditions. The generator is described in Section 5.3.

The generator is written in C# and the code size is about 90kB.

7.1.2 Certification Components

There are three possible approaches to theorem proving and proof checking. XMS does not favour any but currently uses the first one.

1. A tactical theorem prover (*Isabelle, Coq*) can be used for proof construction and proof validation. Proofs are concise and in many cases can be constructed automatically without any manual guidance. However, the prover must be present at Code Consumer side. Such requirement can be the major disadvantage of this approach since the theorem prover is not easily verifiable and even a single inconsistency in the prover engine could lead to incorrect judgements.

2. Proofs can be encoded in a metalogic LF ([38]). This results in long and detailed proofs but the proof checking procedure is cheap at the Code Consumer side. Metalogic proof checkers are short and thus reliable. Additional techniques can be used to shorten proofs ([34]).

3. A logical interpreter can be used as a proof checker ([32]). Such interpreter uses information about the proof structure provided by the Code Producer but instead of recreating the proof it actually checks if the proof exists at all.

7.2 Private Computation

One of free benefits of conforming to static verification with predicates/proofs as certificates is the possibility of using XMS for **Private Computation**.

Suppose that a party **A** needs expensive computation to be performed on some private data. **A** is unable to perform the computation locally. Suppose that party **B** is able to perform the computation for **A**.

However, **A** does not want its private data to be revealed to **B** and **B** does not want its private algorithm to be revealed to **A**.

Using XMS as a certification framework and ASP.NET Web Services as remote computation layer, **A** and **B** can rely on following **XMS Private Computation Protocol**:

1. **A** and **B** ask a trusted party, **C**, to make a Web Service, **W**, available to both of them.

2. **B** publishes its service on **W** together with XMS specification and certificates.

3. **A** asks **W** for the specification of **B**'s service, checks if the specification meets his/her requirements and asks **W** to verify that **B**'s service is correct with respect to its specification using XMS Protocol.

4. **W** verifies the **B**'s service and sends the verification result to **A**.

5. **A** checks the verification status and if it is positive, sends its data to **W** and collects the results of the computation.

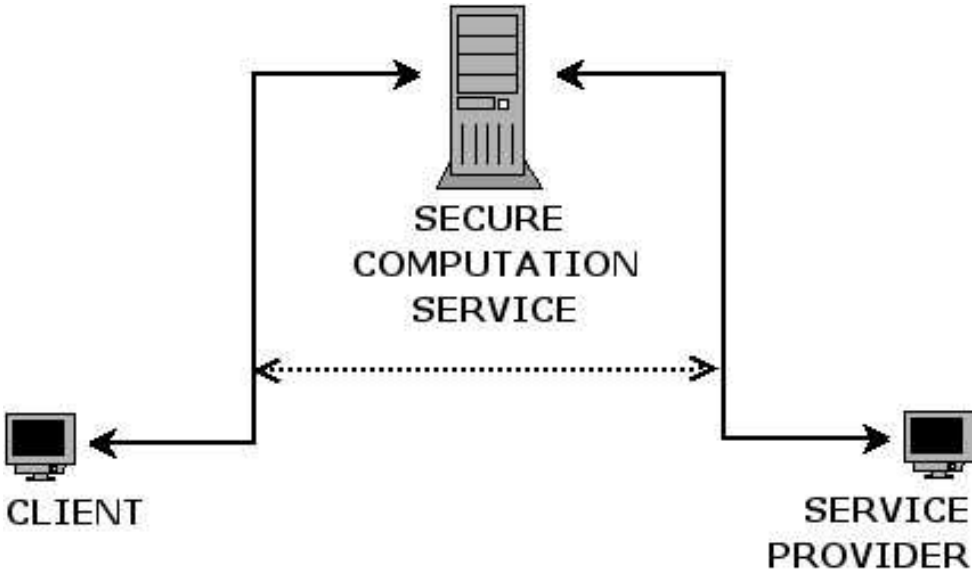

Figure 7.2: XMS Secure Computation Scheme

Chapter 8

Conclusion and Future Work

8.1 Contribution

While XMS is not the first approach to the Design By Contract for an enterprise development platform, it is probably the first one based on two strong paradigms, the Design By Contract and the Proof-Carrying Code. The framework is built for the Microsoft .NET Framework platform and it starts by defining the model and the semantics of the language and then shows how the dynamic and static verification engines form a coherent and complete environment for security policy validation.

Although the framework is built for the Microsoft Intermediate Language, the intermediate language of the .NET platform, we show that it is possible to lift it to enterprise high-level languages to a quite reasonable extent which makes it possible to certify significant parts of software systems. We also provide the Private Computation Protocol which shows that the framework can be used when both the data and an algorithm from two parties should remain private.

The XMS Framework has been implemented and will be released to the community together with this dissertation.

8.2 Future Work

Formal certificates can rely on other certification paradigms like the Model Carrying Code ([37]) where the certificate takes the form of an abstract model of the code execution and model checking techniques are involved to verify these models. The XMS could ultimately unify various approaches. The combination of PCC and MCC seems especially promising.

Three main directions of future XMS development are:

- **support for more MSIL instructions and built-in predicates (Static Verification)**: Currently the static verification does not support all MSIL instructions, for example it does not handle generics. Whether or not these opcodes are liable to static analysis is another research goal. Some built-in predicates could also be supported, such as `ISNULL`.
- **other code instrumentation techniques (Dynamic Verification)**: Although context-bound objects are an easy way to perform code instrumentation, using .NET Profiler API could make the dynamic verification faster and more transparent. The dynamic analysis could be easily turned on/off at the client-side.

- **better integration with high-level languages**: Current handling of loop invariants requires high-level languages to cope with Standard Certificate Specification. Also the developer must be aware of some translation schemes and specific issues regarding the set of active optimizations. This could be too restrictive for some high-level languages, for example functional languages with compilation schemes which are much more indirect. A long-term goal would be to integrate XMS with such languages using one integration strategies we have proposed.

  Yet another issue is the tendency to extend high-level languages with new and new syntactic-sugar. For example, the C# 2.0 language introduces a non-trivial extension for writing custom enumerations using `yield` instruction. The short and concise enumerator:

```
public class Tree
{
    public Tree left;
    public Tree right;
    public int value;

    public Tree( Tree left, Tree right, int value )
    {
        this.left = left;
        this.right = right;
        this.value = value;
    }

    public IEnumerator<int> GetEnumerator()
    {
        if ( left != null )
        {
            IEnumerator<int> e = left.GetEnumerator();
            while ( e.MoveNext() )
                yield return e.Current;
        }

        yield return value;

        if ( right != null )
        {
            IEnumerator<int> e = right.GetEnumerator();
            while ( e.MoveNext() )
                yield return e.Current;
        }
    }
}
```

  translates to a private, internal class implementing the `IEnumerable` interface which handles the enumeration by a compiler-implemented method involving states corresponding

to each `yield` used in the original enumeration definition. Unfortunately, each new language construct brings separate issues to the ”lifting” of certificates.

- **other Safety Policies**: Contracts Safety Policy is not the only interesting Safety Policy that can be verified in a XMS manner. Other formal policies such as Temporal Specifications ([5]) or Non-Interference ([14], [23]) could be adapted to XMS certification scheme. The latter is especially promising since the proof system for symbolic version of the calculi exists ([19]) and could be used in the certification framework.

Bibliography

[1] Andrew W. Appel. Foundational Proof-Carrying Code. *Logic in Computer Science*, 2001.

[2] K. Apt and E. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1997.

[3] Dave Arnold and Jean-Paul Corriveau. Using the .NET Profiler API to Collect Object Instances for Constraint Evaluation. *.NET Technologies 2006, Full Paper Proceedings*, 2006.

[4] Kent Beck. *Test Driven Development: By Example*. Addison-Wesly Longman, 2002.

[5] Andrew Bernard and Peter Lee. Temporal Logic for Proof-Carying Code. *Technical Report, CMU-CS-02-130*, 2002.

[6] Christopher Colby, Peter Lee, and George C. Necula. A Proof-Carrying Code Architecture for Java. *Computer Aided Verification*, pages 557–560, 2000.

[7] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A Certifying Compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, 2000.

[8] D. Kozen D. Harel and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[9] Christian W. Damus. Implementing Model Integrity in EMF with MDT OCL. *IBM Rational Software Resources*, 2007.

[10] Bruce Eckel and Larry O'Brien. *Thinking in C#, Release Candidate*. Prentice Hall, 2002.

[11] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[12] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, 2000.

[13] Amy Felty and Andrew W. Appel. Semantic Model of Types and Machine Instructions for Proof-Carying Code. *Symposium on Principles of Programming Languages*, 2000.

[14] Riccardo Focardi and Roberto Gorrieri. Classification of Security Properties. *The International School on Foundations of Security Analysis and Design (FOSAD)*, 2000.

[15] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[16] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. *ACM SIGPLAN Notices*, 36(3):248–260, 2001.

[17] Scott Hazelhurst and Carl-Johan H. Seger. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Formal Methods in Systems Design*, 1995.

[18] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 1969.

[19] Anna Ingólfsdóttir and Huimin Lin. A Symbolic Approach to Value-Passing Processes. 2000.

[20] Newkirk JW. and Vorontsov AA. *Test-Driven Development in Microsoft .NET*. Microsoft Press, 2004.

[21] Joshua Kerievsky. *Refactoring To Patterns*. Addison-Wesley, 2004.

[22] Gregor Kiczales et al. Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming*, 1997.

[23] H. Lin. Symbolic Bisimulations and Proof Systems for the π-Calculus. *Technical Report 1994:07, University of Sussex*, 1994.

[24] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[25] Bertrand Meyer. Applying "Design by Contract" . Computer, *IEEE, Volume 25, Issue 10*, 1992.

[26] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice Hall, 1997.

[27] Microsoft. Common Language Infrastructure Specification. *ECMA-335 Specification*, 2002.

[28] M. Leino Mike Barnett, K. Rustan and Wolfram Schulte. The Spec# Programming System: An Overview. *CASSIS*, 2004.

[29] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge Univerisy Press, 1999.

[30] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[31] David Evans Nathanael Paul. Comparing Java and .NET Security: Lessons Learned and Missed. *Technical Report, University of Virginia*, 2004.

[32] George C. Necula and S. P. Rahul. Oracle-Based Checking of Untrusted Software. *Symposium on Principles of Programming Languages*, 2001.

[33] George Ciprian Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.

[34] George Ciprian Necula and Peter Lee. Efficient Representation and Validation of Logical Proofs. *Technical Report, CMU-CS-97-172*, 1997.

[35] Cees Pierik and Frank S. de Boer. A Syntax-Directed Hoare Logic for Object-Oriented Programming Contepts. *Technical Report UU-CS-2003-010*, 2003.

[36] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. *European Symosium un Programming (ESOP '99)*, 1576:162–176, 1999.

[37] Samik Basu Sandeep Bhatkar Daniel C. DuVarney R. Sekar, V.N. Venkatakrishnan. Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications. *Technical Report Stony Brook University*, 2003.

[38] Furio Honsell Robert Harper and Gordon Plotkin. A Framework for Defining Logics. *Logic in Computer Science*, 1987.

[39] Peter Y. A. Ryan. Mathematical models of computer security. *The International School on Foundations of Security Analysis and Design (FOSAD)*, pages 1–62, 2000.

[40] Grigoryev D. Maslennikov A. Safonov V., Gratchev M. Aspect.NET, Aspect-Oriented Toolkit for Microsoft .NET. *Proceedings of the .NET Technologies 2006 Conference*, 2006.

[41] Ken Schwaber. *Agile Project Management with Scrum*. Microsoft Press, 2004.

[42] Sorin Serban. Parsing the IL of a Method Body. *http://www.codeproject.com*, 2006.

[43] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. *Symposium on Principles of Programming Languages*, pages 149–160, 1998.

[44] Michael S. V. Turner. *Microsoft Solutions Framework Essentials*. Microsoft Press, 2006.

[45] David von Oheimb. Hoare Logic for Java in Isabelle/HOL. *Concurrency - Practice and Experience*, 2001.

[46] Wiktor Zychla. eXtensible Multi Security, Contracts for .NET Platform. *Journal of .NET Technologies*, 4, 2006.

List of Figures

1.1 XMS safety versus PCC type-safety 5

2.1 Comparison of static and dynamic techniques 9

3.1 Overview of basic PCC protocol 13
3.2 A modification of PCC protocol for the XMS 15

4.1 Structure of a managed assembly 18
4.2 High-level compilers versus JIT compilers 19
4.3 Valid and verifiable IL 20
4.4 .NET Framework Code-based policy configuration tool 23
4.5 Naming conventions 24
4.6 Types 25
4.7 Field and method signatures 25
4.8 Basic inheritance rules 25
4.9 Selected IL Control Flow Instructions 26
4.10 IL Arithmetical Instruction Set 27
4.11 Selected IL instructions for fields, arguments and local variables 27
4.12 IL Instructions for Calling Methods 28
4.13 Selected IL Instructions for Class Manipulation 29
4.14 IL Instructions for Vector Operations 29
4.15 Result values 30
4.16 Local memory context 31
4.17 Semantics of selected Control Flow Instructions 34
4.18 Semantics of Arithmetical Instructions 35
4.19 Semantics of Arithmetical Instructions, cont. 36
4.20 Semantics of Instructions for Addressing Fields, Arguments and Local Variables 37
4.21 Semantics of Instructions for Calling Methods 37
4.22 Semantics of Instructions for Addressing Objects 38
4.23 Semantics of Instructions for Vector Operations 38

5.1 Definition of Verification Condition for Design By Contract 55

6.1 Translation scheme for `for` instruction 74

7.1 XMS Framework at the Code-Producer side 82
7.2 XMS Secure Computation Scheme 84

List of Tables

5.1 XMS Symbolic Evaluator for Design By Contract 62

A.1 IL Instruction Set 97

Appendix A

MSIL Instruction Set

The table below contains full MSIL 1.0 instruction set. Each instruction is followed by a symbol which shows its status from the VCGen point of view:

- ● - the opcode is fully supported by the VCGen
- ⊘ - the opcode is skipped by the VCGen
- ○ - the opcode is not supported by the VCGen

Notice that the IL instruction set is optimized to produce as small assemblies as possible. This is why most frequent instructions have their own opcodes (for example the `ldc.i4.0` to `ldc.i4.8` load 32-bit integer values 0 to 8 onto the stack) and the most rare instructions have two-byte opcodes (for example the `ldarg` instruction with 32-bit parameter).

Table A.1: IL Instruction Set

Code	Instruction	Supported	Comments
00	`nop`	●	
01	`break`	⊘	used for debugging purposes
02	`ldarg.0`	●	
03	`ldarg.1`	●	
04	`ldarg.2`	●	
05	`ldarg.3`	●	
06	`ldloc.0`	●	
07	`ldloc.1`	●	
08	`ldloc.2`	●	
09	`ldloc.3`	●	
0A	`stloc.0`	●	
0B	`stloc.1`	●	
0C	`stloc.2`	●	
0D	`stloc.3`	●	
0E	`ldarg.s`	●	
0F	`ldarga.s`	●	
10	`starg.s`	●	
11	`ldloc.s`	●	
12	`ldloca.s`	●	

Code	Instruction	Supported	Comments
13	stloc.s	●	
14	ldnull	●	
15	ldc.i4.M1	●	
16	ldc.i4.0	●	
17	ldc.i4.1	●	
18	ldc.i4.2	●	
19	ldc.i4.3	●	
1A	ldc.i4.4	●	
1B	ldc.i4.5	●	
1C	ldc.i4.6	●	
1D	ldc.i4.7	●	
1E	ldc.i4.8	●	
1F	ldc.i4.s	●	
20	ldc.i4	●	
21	ldc.i8	●	
22	ldc.r4	●	
23	ldc.r8	●	
25	dup	●	
26	pop	●	
27	jmp	●	
28	call	●	
29	calli	○	unverifiable instruction for indirect calls
2A	ret	●	
2B	br.s	●	
2C	brfalse.s	●	
2D	brtrue.s	●	
2E	beq.s	●	
2F	bge.s	●	
30	bgt.s	●	
31	ble.s	●	
32	blt.s	●	
33	bne.un.s	●	
34	bge.un.s	●	
35	bgt.un.s	●	
36	ble.un.s	●	
37	blt.un.s	●	
38	br	●	
39	brfalse	●	
3A	brtrue	●	
3B	beq	●	
3C	bge	●	
3D	bgt	●	
3E	ble	●	
3F	blt	●	
40	bne.un	●	
41	bge.un	●	
42	bgt.un	●	
43	ble.un	●	

Code	Instruction	Supported	Comments
44	blt.un	●	
45	switch	●	
46	ldind.i1	●	
47	ldind.u1	●	
48	ldind.i2	●	
49	ldind.u2	●	
4A	ldind.i4	●	
4B	ldind.u4	●	
4C	ldind.i8	●	
4D	ldind.i	●	
4E	ldind.r4	●	
4F	ldind.r8	●	
50	ldind.ref	●	
51	stind.ref	●	
52	stind.i1	●	
53	stind.i2	●	
54	stind.i4	●	
55	stind.i8	●	
56	stind.r4	●	
57	stind.r8	●	
58	add	●	
59	sub	●	
5A	mul	●	
5B	div	●	
5C	div.un	●	
5D	rem	●	
5E	rem.un	●	
5F	and	●	
60	or	●	
61	xor	●	
62	shl	●	
63	shr	●	
64	shr.un	●	
65	neg	●	
66	not	●	
67	conv.i1	⊘	
68	conv.i2	⊘	
69	conv.i4	⊘	
6A	conv.i8	⊘	
6B	conv.r4	⊘	
6C	conv.r8	⊘	
6D	conv.u4	⊘	
6E	conv.u8	⊘	
6F	callvirt	●	
70	cpobj	○	should be handled like ref types
71	ldobj	○	should be handled like ref types
72	ldstr	●	
73	newobj	●	

Code	Instruction	Supported	Comments
74	castclass	⊘	
75	isinst	○	not possible to verify statically
76	conv.r.un	●	
79	unbox	●	
7A	throw	●	
7B	ldfld	●	
7C	ldflda	○	
7D	stfld	●	
7E	ldsfld	●	
7F	ldsflda	○	
80	stsfld	●	
81	stobj	○	
82	conv.ovf.i1.un	⊘	
83	conv.ovf.i2.un	⊘	
84	conv.ovf.i4.un	⊘	
85	conv.ovf.i8.un	⊘	
86	conv.ovf.u1.un	⊘	
87	conv.ovf.u2.un	⊘	
88	conv.ovf.u4.un	⊘	
89	conv.ovf.u8.un	⊘	
8A	conv.ovf.i.un	⊘	
8B	conv.ovf.u.un	⊘	
8C	box	○	
8D	newarr	●	
8E	ldlen	●	
8F	ldelema	●	
90	ldelem.i1	●	
91	ldelem.u1	●	
92	ldelem.i2	●	
93	ldelem.u2	●	
94	ldelem.i4	●	
95	ldelem.u4	●	
96	ldelem.i8	●	
97	ldelem.i	●	
98	ldelem.r4	●	
99	ldelem.r8	●	
9A	ldelem.ref	●	
9B	stelem.i	●	
9C	stelem.i1	●	
9D	stelem.i2	●	
9E	stelem.i4	●	
9F	stelem.i8	●	
A0	stelem.r4	●	
A1	stelem.r8	●	
A2	stelem.ref	●	
B3	conv.ovf.i1	⊘	
B4	conv.ovf.u1	⊘	
B5	conv.ovf.i2	⊘	

Code	Instruction	Supported	Comments
B6	conv.ovf.u2	⊘	
B7	conv.ovf.i4	⊘	
B8	conv.ovf.u4	⊘	
B9	conv.ovf.i8	⊘	
BA	conv.ovf.u8	⊘	
C2	refanyval	○	
C3	ckfinite	○	
C6	mkrefany	○	
D0	ldtoken	○	
D1	conv.u2	⊘	
D2	conv.u1	⊘	
D3	conv.i	⊘	
D4	conv.ovf.i	⊘	
D5	conv.ovf.u	⊘	
D6	add.ovf	●	
D7	add.ovf.un	●	
D8	mul.ovf	●	
D9	mul.ovf.un	●	
DA	sub.ovf	●	
DB	sub.ovf.un	●	
DC	endfinally	●	
DD	leave	●	
DE	leave.s	●	
DF	stind.i	●	
E0	conv.u	●	
FE 00	arglist	○	
FE 01	ceq	●	
FE 02	cgt	●	
FE 03	cgt.un	●	
FE 04	clt	●	
FE 05	clt.un	●	
FE 06	ldftn	●	
FE 07	ldvirtftn	○	
FE 09	ldarg	●	
FE 0A	ldarga	○	
FE 0B	starg	●	
FE 0C	ldloc	●	
FE 0D	ldloca	○	
FE 0E	stloc	●	
FE 0F	localloc	○	
FE 11	endfilter	○	
FE 12	unaligned.	○	
FE 13	volatile.	○	
FE 14	tail.	○	
FE 15	initobj	○	
FE 17	cpblk	○	
FE 18	initblk	○	
FE 1A	rethrow	○	

Code	Instruction	Supported	Comments
FE 1C	`sizeof`	○	
FE 1D	`refanytype`	○	

Appendix B

The Soundness Theorem

This appendix contains proof of soundness for the Safety Theorem 5.1: The proof technique of this theorem is adopted from the original proof presented in [33]. What we are left to do is to adopt this technique to this particular case - the Microsoft Intermediate Language and the Design By Contract policy.

Theorem B.1 (Soundness of Verification Condition Generator for Design by Contract)**.** *If the verification condition for a given module M is valid, i.e.* $\models VC(\mathcal{M})$ *then all executions of any module methods are correct with respect to the Design By Contract security policy, i.e.* $Safe_{SC}(\mathcal{M})$.

Since the verification condition of a module is the conjunction of conditions of all methods from the module, we are left to prove the theorem for any single method $F \in \mathcal{M}$.

The proof uses the induction on execution length of a method body. From the definition of the safety policy we can assume that the execution was started in a state that satisfies the precondition. Then at each state of the runtime environment we show that there exists a corresponding state of the symbolic evaluator. This correspondence is the main concept of the proof.

To be able to use induction we first state the **induction hypothesis** for an execution state $\Sigma = (i, \rho)$ and corresponding symbolic evaluator state $SE(i, \sigma, \mathcal{L})$. The induction hypothesis reflects the fact that:

- the verification condition for F is valid, i.e. $\models VC(F)$
- the execution was initiated in a state satisfying the precondition, i.e. $\models Pre_F$
- the state Σ of the execution of F and the state $SE(i, \sigma, \mathcal{L})$ of the symbolic evaluator are related

Definition B.1. *Let us consider the simultaneous execution of the MSIL Runtime Environment executing the method* F *in a state* (i, ρ) *and of the symbolic evaluator in the state* $SE(i, \sigma, \mathcal{L})$. *Let* τ *be the current evaluation of symbolic values. We say that the induction hypothesis holds in this state (we write* $\models IH_F(i, \sigma, \mathcal{L}, \tau, \rho)$*) when:*

1. *the current evaluation of the verification condition is valid, i.e.* $\models \tau(SE_F(i, \sigma, \mathcal{L}))$
2. σ, τ *and* ρ *are related such that* $\models \tau(\sigma(v)) = \rho(v)$ *for any v in Dom(*ρ*)*
3. τ *correctly captures the invariant context* $\mathcal{L}$, *i.e. if* $\mathcal{L} = \mathcal{L}_1 + (i, \sigma'_1)$ *where*

(a) $\mathcal{L}$ *is empty and for all* v *we have* $\models \tau(\sigma_0(v)) = \rho_0(v)$*, or*

(b) $\mathcal{L} = \mathcal{L}_1 + (i, \sigma'_1)$ *with* $\sigma'_1 = \sigma_1[\ldots, l_V(v_i \mapsto v'_i{}^{v_i \in Var}), u'_1 \ldots u'_k \cdot s]$ *then*

i. $F_i = Inv(P, Var, k)$

ii. v'_i *and* u'_i *are fresh variables*

iii. $\tau = \tau_1 + [v'_i \mapsto t_i, u'_i \mapsto t'_i]$

iv. $\models \tau_1(\sigma_1(P))$

v. $\models \tau_1(\forall v'_0, \ldots, v'_m . u'_1 \ldots u'_k . \sigma'_1(P) \Rightarrow SE(i, \sigma', \ldots))$

vi. τ_1 *is correct with respect to* $\mathcal{L}_1$

Having stated the induction hypothesis we will show that:

- the hypothesis holds at the time of invocation of F
- the hypothesis holds at each step of the execution progress of F
- methods are safely invoked from within F
- the hypothesis implies that at the end of the execution F is secure with respect to the safety policy, i.e. $\models Post_F$

The lemma below restates the main theorem by resolving the notion of safety using the definition of the safety policy (Definition 5.1) and the induction hypothesis.

Lemma B.1. *If the verification condition for given method* F *is valid, i.e.* $\models VC(F)$ *then for any initial state* $(0, \rho_0)$ *such that* $Pre_F(\rho_0)$ *and assuming that all invoked methods are safe we have that for any reachable state* $\Sigma = (i, \rho)$ *there exists* σ, $\mathcal{L}$ *and* τ *such that the induction hypothesis holds, i.e.* $IH_F(i, \sigma, \mathcal{L}, \tau, \rho)$ *and we have that either::*

- $F_i =$ `ret` *and* $Post_F(\rho)$
- $F_i \neq$ `ret`*, there exists* Σ' *such that* $\Sigma \mapsto \Sigma'$ *and the induction hypothesis holds in the new state* Σ'

The lemma B.1 can be further reformulated to four other auxiliary lemmas.

Lemma B.2 (Invocation)**.** *If the verification condition for* F *holds, i.e.* $\models VC(F)$ *and* F *is invoked in a "safe" state, i.e.* $Pre_F(\rho_0)$ *then there exists* σ_0 *and* τ_0 *such that* $IH(i, \sigma_0, \epsilon, \tau_0, \rho_0)$*.*

Lemma B.3 (Progress)**.** *If the induction hypothesis holds in a state* (i, ρ)*, i.e.* $\models IH(i, \sigma, \mathcal{L}, \tau, \rho)$ *and the current instruction is* **not** *a call and* **not** *a return instruction then when the execution engine makes a progress to the state* (j, ρ') *then there exists* σ'*,* $\mathcal{L}'$ *and* τ' *such that* $\models IH(j, \sigma', \mathcal{L}', \tau', \rho')$*.*

Lemma B.4 (Call)**.** *If the induction hypothesis holds in a state* (i, ρ)*, i.e.* $\models IH(i, \sigma, \mathcal{L}, \tau, \rho)$ *and the current instruction* **is** *the call of a method* G *such that* $Safe(G)$ *then if the execution engine returns from the call to the state* $(i + 1, \rho')$ *then there exists* σ' *and* τ' *such that* $\models IH(i + 1, \sigma', \mathcal{L}, \tau', \rho')$*.*

Lemma B.5 (Return)**.** *If the induction hypothesis holds in a state* (i, ρ)*, i.e.* $\models IH(i, \sigma, \mathcal{L}, \tau, \rho)$ *and the current instruction* **is** *the return instruction then the return is safe, i.e.* $Post_F(\rho)$*.*

Lemma B.2, allows us to establish the induction hypothesis at the method invocation, lemma B.3, guarantees that if the execution makes progress then the induction hypothesis holds in the new state, lemma B.4, states that the method call preserves the induction hypothesis and the last one, B.5, deals with the method return case.

We also state the congruence lemma which will be used throughout proofs of B.2 to B.5. The lemma can be easily proven by the structural induction over P.

Lemma B.6 (Congruence)**.** *If P is a predicate, ρ is a state of the execution engine, σ is the state of the symbolic evaluator and τ is the evaluation of symbolic values from σ to values such that $\models \tau(\sigma(v)) = \rho(v)$, for any v in Dom(ρ), then $\models \sigma(\tau(P))$ if and only if $\models (\rho(P))$.*

Proof of Invocation Lemma B.2 Since the verification condition is valid for F we have $\forall a_0, \ldots, a_n.\sigma_0^F(Pre_F) \Rightarrow SE(0, \sigma_0^F, \emptyset)$ where $\sigma_0^F = (l_A(a_i \mapsto a_i), l_V(v_i \mapsto 0), \epsilon)$.

Since the values in local argument vector are universally quantified, we have $\models \tau_0(\sigma_0^F(Pre_F) \Rightarrow SE(0, \sigma_0^F, \emptyset))$ where $\tau_0 = [a_i \mapsto \rho(a_i)]$ (because we drop the universal quantification in favor of fixed values).

What we have to prove is that the induction hypothesis is true, i.e. $\models IH(i, \sigma_0, \epsilon, \tau_0, \rho_0)$.

The clause 2 is just by the definition of σ_0 and τ_0. By the assumption $Pre_F(\rho_0)$ and clause 2 we have that $\tau_0(\sigma_0(Pre_F))$ that gives us the conclusion by using the modus pones.

□

Proof of Progress Lemma B.3 To show that the induction hypothesis holds when the runtime environment makes a progress in the execution we analyze the last instruction used for symbolic evaluation. We will prove this lemma for few example instructions.

invariant case - first time We have to prove the invariant $IH_F(i, \sigma', \mathcal{L}, \tau', \rho')$ where

$$
\begin{array}{ll}
u_i' & \text{are new variables so that} \\
 & \sigma(u_i') = u_i \text{ and } \tau(\sigma(u_i')) = \rho(u_i') \\
v_i' & \text{are new variables so that} \\
 & \sigma' = \sigma[v_i \mapsto v_i', u_1', \ldots, u_k' \cdot s] \\
 & \tau' = \tau + [v_i' \mapsto \rho(v_i)]
\end{array}
$$

Note that because of the way the τ' is defined we have that $\tau'(\sigma'(v)) = \rho(v)$.

In order to show that the clause 2 of the induction hypothesis holds we have to refer to the definition of the VCGen. By the definition of the VCGen for this case we have:

$$
\begin{array}{l}
\models \tau(\sigma(P)) \\
\models \tau'(\sigma'(P)) \implies \tau'(SE(i, \sigma', \mathcal{L} + (i, \sigma')))
\end{array}
$$

Using the congruence lemma we get from the first clause that $\models \rho(P)$ and then because $\tau'(\sigma'(v)) = \rho(v)$ we have that $\models \tau'(\sigma'(P))$. From this and from the second clause above we have the induction hypothesis.

invariant case - second time The proof of this clause is a reformulation of the corresponding clause from [33].

case `ldarg` v, `starg` v The only non-trivial clause of the induction hypothesis is clause 1.

In case of the `ldarg` v instruction we have to show that $\tau(l_A(v)) = \rho(l_A(v))$ which is true by the induction hypothesis.

In case of the `starg` v instruction we have to show that $\tau(l_A[v \mapsto u]) = \rho(l_A[v \mapsto u])$. But since $l_A[v \mapsto u](v') = l_A(v')$ for $v \neq v'$ we only have to check that $\tau(l_A[v \mapsto u])(u) = \rho(l_A[v \mapsto u](u))$ since the equality is true in case of any other value because of the induction hypothesis.

We also have $\tau(l_A[v \mapsto u])(u) = \tau(u) = \rho(u) = \rho(l_A[v \mapsto u](u))$ again because of the induction hypothesis.

case `brtrue` l The clause 2 is trivial in this case because both the runtime environment and the symbolic evaluator just pop the value from the stack. The new state of the runtime environment is ρ' and new state of the symbolic evaluator is σ'.

To prove the clause 1 we have to analyze two cases depending of the current value v at the top of the stack.

If $\rho(v) = 0$ then the next state is $(i + 1, \rho')$ so we have to prove that the clause 1 holds in the new state, i.e. $\models \tau(SE(i + 1, \rho'))$.

Since the verification condition for current instruction holds and it is the conjunction of two cases then both cases hold.

Specifically this means that $\models \tau(\sigma(v) = 0 \Rightarrow SE(i + 1, \rho'))$. But since $\rho(v) = 0$ and clause 2 we have that $\models \tau(SE(i + 1, \rho'))$.

□

Proof of Call Lemma B.4

The structure of this clause is compatible with the corresponding lemma from [33]. What we have to show is that the verification is also correct for polymorphic calls.

Since both the induction hypothesis and *VCI* part of the Verification Condition hold, we have:

$$\sigma^G(Pre_G) \wedge \forall u.\sigma^G(Post_G[u/VALUE]) \Rightarrow SE(i + 1, \sigma', \mathcal{L})$$
$$Pre_G \implies Pre_{Inherited(G)} \wedge Post_{Inherited(G)} \implies Post_G$$

From this we conclude that

$$\sigma^G(Pre_{Inherited(G)}) \wedge \forall u.\sigma^G(Post_{Inherited(G)}[u/VALUE]) \Rightarrow SE(i + 1, \sigma', \mathcal{L})$$

This means that if the induction hypothesis holds for a base class call then it also holds for any inherited class call.

□

Proof of Return Lemma B.5 From the clause 1 of the induction hypothesis and from the definition of the symbolic evaluator we have that $\models \tau(\sigma(Post_F))$. From the clause 2 of the induction hypothesis we have that $\models \rho(Post_F)$ which is what we need to prove.

□

Appendix C

Examples

C.1 Dynamic Verification Engine

The Dynamic Verification Engine uses code instrumentation to supervise the execution and evaluate specificatin predicates at the run time. Detailed description of the engine can be found in Section 5.2 on page 44.

The example test suite class:

```
/* XMSIntercept attribute required by dynamic engine */
[XMSIntercept]
public class TestSuite1 : ContextBoundObject
{
  /// <summary>
  /// Sum of provded parameters.
  /// </summary>
  /// <param name="x"></param>
  /// <param name="y"></param>
  /// <returns></returns>
  [Process(typeof(XMSProcessor))]
  [XMS_Spec(
    "x >= 0 && y >= 0", /* precondition */
    "VALUE == x+y",     /* postcondition */
    ""                  /* invariants */
    )]
  public int TestInt1( int x, int y )
  {
    return x+y;
  }

  /// <summary>
  /// Postcondition refers to VALUE returned by the method.
  /// </summary>
  /// <param name="a"></param>
  /// <returns></returns>
  [Process(typeof(XMSProcessor))]
  [XMS_Spec( "notnull(a)", "VALUE == 1", "" )]
```

```
  public int TestArrayListCount( ArrayList a )
  {
    a.Add(1);
    return a.Count;
  }

  /// <summary>
  /// Example of builtin notnull(*) predicate.
  /// </summary>
  /// <returns></returns>
  [Process( typeof( XMSProcessor ) )]
  [XMS_Spec( "true", "notnull(a)", "" )]
  public ArrayList FactoryMethod()
  {
    ArrayList a = new ArrayList();

    return a;
  }

  /// <summary>
  /// Precondition checks if valid array is provided.
  /// </summary>
  /// <param name="a"></param>
  [Process(typeof(XMSProcessor))]
  [XMS_Spec( "notnull(a)", "true", "" )]
  public void TestArray_1( int[] a )
  {
    return;
  }

  /// <summary>
  /// Postcondition can refer to original values of parameters.
  /// </summary>
  /// <param name="a"></param>
  [Process( typeof( XMSProcessor ) )]
  [XMS_Spec( "true", "x == y_ORIGINAL && y == x_ORIGINAL", "" )]
  public void Swap( ref int x, ref int y )
  {
    int z = x;
    x = y;
    y = z;
  }
}
```

The tests are executed from following context:

```
TestSuite1 ts = new TestSuite1();

/* this should pass */
int int1 = ts.TestInt1( 5, 6 );
```

```
/* this should fail ( precondition ) */
int array = ts.TestInt1( -5, 6 );

ArrayList arraylist = new ArrayList();
/* this should pass */
int count0 = ts.TestArrayListCount( arraylist );
arraylist.Add( 1 );
/* this shoudl fail ( postcondition ) */
int count1 = ts.TestArrayListCount( arraylist );

/* this should pass */
ArrayList arrayList = ts.FactoryMethod();

int[] x = new int[5]; x[0] = 0;
ts.TestArray_1( x );

int u = 0, v = 1;
/* this should pass */
ts.Swap( ref u, ref v );
```

Actual output of the XMS dynamic engine:

```
-----------------------------
Preprocessing TestSuite1.TestInt1.
  specification:
  Pre=[x >= 0 && y >= 0]
  Post=[VALUE == x+y]
  Invs=[]


Precondition : (x >= 0) && (y >= 0)
  Substituted expression : (5 >= 0) && (6 >= 0)
  Evaluated expression : True

Postcondition : VALUE == (x + y)
  Substituted expression : 11 == (5 + 6)
  Evaluated expression : True

-----------------------------
Preprocessing TestSuite1.TestInt1.
  specification:
  Pre=[x >= 0 && y >= 0]
  Post=[VALUE == x+y]
  Invs=[]


Precondition : (x >= 0) && (y >= 0)
  Substituted expression : (-5 >= 0) && (6 >= 0)
  Evaluated expression : False
```

```
Postcondition : VALUE == (x + y)
  Substituted expression : 1 == (-5 + 6)
  Evaluated expression : True
Testing failed for method TestInt1.

------------------------------
Preprocessing TestSuite1.TestArrayListCount.
  specification:
  Pre=[true]
  Post=[VALUE == 1]
  Invs=[]


Precondition : true
  Substituted expression : true
  Evaluated expression : True

Postcondition : VALUE == 1
  Substituted expression : 1 == 1
  Evaluated expression : True

------------------------------
Preprocessing TestSuite1.TestArrayListCount.
  specification:
  Pre=[true]
  Post=[VALUE == 1]
  Invs=[]


Precondition : true
  Substituted expression : true
  Evaluated expression : True

Postcondition : VALUE == 1
  Substituted expression : 3 == 1
  Evaluated expression : False
Testing failed for method TestArrayListCount.

------------------------------
Preprocessing TestSuite1.FactoryMethod.
  specification:
  Pre=[true]
  Post=[notnull(a)]
  Invs=[]


Precondition : true
  Substituted expression : true
  Evaluated expression : True
```

```
Postcondition : notnull(a)
  Substituted expression : true
  Evaluated expression : True

------------------------------
Preprocessing TestSuite1.TestArray_1.
  specification:
  Pre=[notnull(a)]
  Post=[true]
  Invs=[]


Precondition : notnull(a)
  Substituted expression : true
  Evaluated expression : True

Postcondition : true
  Substituted expression : true
  Evaluated expression : True

------------------------------
Preprocessing TestSuite1.Swap.
  specification:
  Pre=[true]
  Post=[x == y_ORIGINAL && y == x_ORIGINAL]
  Invs=[]


Precondition : true
  Substituted expression : true
  Evaluated expression : True

Postcondition : (x == y_ORIGINAL) && (y == x_ORIGINAL)
  Substituted expression : (1 == 1) && (0 == 0)
  Evaluated expression : True
```

C.2 Static Verification Engine

The Static Verification Engine uses the Verification Condition Generator to analyze assembly metadata, read MSIL instructions and perform symbolic evaluation to produce verification predicates. Detailed description of the engine can be found in Section 5.3 on page 53.

Example C.1 (Constructors) Class constructors are analyzed just like all other methods. Consider following C# code:

```
public class StaticTestsSuite
{
  public int a, b, c;
```

```
  [XMS_Spec( "true", "this.a = av && this.b = bv && this.c = cv", "" )]
  public StaticTestsSuite( int av, int bv, int cv )
  {
    this.a = av;
    this.b = bv;
    this.c = cv;
  }
}
```

The constructor C# code compiled to MSIL follows:

```
.method public hidebysig specialname rtspecialname instance
 void .ctor(int32 av, int32 bv, int32 cv) cil managed
{
 .custom instance
  void [UWr.XMS.Base]UWr.XMS.Base.XMS_Spec::.ctor(string, string, string) =
  { string('true')
    string('this.a = av && this.b = bv && this.c = cv')
    string('') }

  .maxstack 8
  L_0000: ldarg.0 // load "this" reference
  L_0001: call instance void [mscorlib]System.Object::.ctor()
  L_0006: ldarg.0
  L_0007: ldarg.1 // load av and store into this.a
  L_0008: stfld int32 Uwr.XMS.Tests.StaticTestsSuite::a
  L_000d: ldarg.0
  L_000e: ldarg.2 // load bv and store into this.b
  L_000f: stfld int32 Uwr.XMS.Tests.StaticTestsSuite::b
  L_0014: ldarg.0
  L_0015: ldarg.3 // load cv and store into this.c
  L_0016: stfld int32 Uwr.XMS.Tests.StaticTestsSuite::c
  L_001b: ret
}
```

The VCGen produces following verification condition:

```
forall av. forall bv. forall cv. true => (((av = av) && (bv = bv)) && (cv = cv))
```

Example C.2 (Constructor calling) The constructor from the Example C.1 is called from external code and values are provided for constructor parameters.

```
[XMS_Spec( "true", "VALUE = x + x + x", "" )]
public int TestTheConstructor( int x )
{
  StaticTestsSuite ts = new StaticTestsSuite( x, x, x );
  return ts.a + ts.b + ts.c;
}
```

The above code is compiled to MSIL as follows:

```
.method public hidebysig instance int32 TestConstructor(int32 x) cil managed
{
 .custom instance
  void [UWr.XMS.Base]UWr.XMS.Base.XMS_Spec::.ctor(string, string, string) =
  { string('true') string('VALUE = x + x + x') string('') }
  .maxstack 4
  .locals init (
   [0] class Uwr.XMS.Tests.StaticTestsSuite ts)

  L_0000: ldarg.1 // load argument onto the stack
  L_0001: ldarg.1
  L_0002: ldarg.1
  L_0003: newobj instance void Uwr.XMS.Tests.StaticTestsSuite::
                               .ctor(int32, int32, int32)
                  // call the StaticTestsSuite( x, x, x )
  L_0008: stloc.0 // store the reference in the local variable
  L_0009: ldloc.0 // load ts.a
  L_000a: ldfld int32 Uwr.XMS.Tests.StaticTestsSuite::a
  L_000f: ldloc.0 // load ts.b
  L_0010: ldfld int32 Uwr.XMS.Tests.StaticTestsSuite::b
  L_0015: add     // ts.a + ts.b
  L_0016: ldloc.0 // load ts.c
  L_0017: ldfld int32 Uwr.XMS.Tests.StaticTestsSuite::c
  L_001c: add     // ts.a + ts.b + ts.c
  L_001d: ret
}
```

The VCGen produces following verification condition:

```
forall Uwr.XMS.Tests.StaticTestsSuite__1.a.
forall Uwr.XMS.Tests.StaticTestsSuite__1.b.
forall Uwr.XMS.Tests.StaticTestsSuite__1.c.
forall x.
  true =>
  ((((Uwr.XMS.Tests.StaticTestsSuite__1.a = x) &&
     (Uwr.XMS.Tests.StaticTestsSuite__1.b = x)) &&
     (Uwr.XMS.Tests.StaticTestsSuite__1.c = x)) =>
       (((Uwr.XMS.Tests.StaticTestsSuite__1.a +
          Uwr.XMS.Tests.StaticTestsSuite__1.b) +
          Uwr.XMS.Tests.StaticTestsSuite__1.c) =
          ((x + x) + x)))
```

Example C.3 (Method calling) The method from the Example C.2 is called from external code.

```
[XMS_Spec( "true", "VALUE = 6", "" )]
public int MethodCalling()
{
  int val = 1;
```

```
  // since TestConstructor actually return value is
  // 3 times the parameter value,
  // following should return 6
  return 2 * TestConstructor( val );
}
```

The MSIL translation is as follows:

```
.method public hidebysig instance int32 MethodCalling() cil managed
{
    .custom instance
    void [UWr.XMS.Base]UWr.XMS.Base.XMS_Spec::.ctor(string, string, string) =
    { string('true') string('VALUE = 6') string('') }
    .maxstack 3
    .locals init (
        [0] int32 val)
    L_0000: ldc.i4.1  // load 1
    L_0001: stloc.0   // val = 1
    L_0002: ldc.i4.2  // load 2
    L_0003: ldarg.0   // load this
    L_0004: ldloc.0   // load val
    L_0005: call instance int32 Uwr.XMS.Tests.StaticTestsSuite::
            TestConstructor(int32)
    L_000a: mul       // 2 * TestConstructor( val )
    L_000b: ret
}
```

The VCGen produces following predicate:

```
true => (true &
  (forall Int32__1. (Int32__1 = ((1 + 1) + 1)) =>
     ((2 * Int32__1) = 6)))
```

Example C.4 (Symbolic arrays) Consider following C# code:

```
[XMS_Spec( "i != j", "VALUE = 3", "" )]
int Arr_SymbolicIndexes( int[] array, int i, int j )
{
  array[i] = 1;
  array[j] = 2;

  return array[i] + array[j];
}
```

The C# compiler output for this method is as follows:

```
.method private hidebysig instance int32
  Arr_SymbolicIndexes(int32[] array, int32 i, int32 j) cil managed
{
```

```
    .custom instance
    void [UWr.XMS.Base]UWr.XMS.Base.XMS_Spec::.ctor(string, string, string) =
    { string('true') string('VALUE = 3') string('') }
    .maxstack 8
    L_0000: ldarg.1   \\ load array
    L_0001: ldarg.2   \\ load i
    L_0002: ldc.i4.1  \\ load 1
    L_0003: stelem.i4 \\ array[i] = 1
    L_0004: ldarg.1   \\ load array
    L_0005: ldarg.3   \\ load j
    L_0006: ldc.i4.2  \\ load 2
    L_0007: stelem.i4 \\ array[i] = 2
    L_0008: ldarg.1
    L_0009: ldarg.2
    L_000a: ldelem.i4 \\ load array[i] (=1)
    L_000b: ldarg.1
    L_000c: ldarg.3
    L_000d: ldelem.i4 \\ load array[j] (=2)
    L_000e: add
    L_000f: ret       \\ ret array[i] + array[j]
}
```

The VCGen produces following verification predicate:

```
forall array[i]. forall i. forall j.
  i != j =>
  (
   ((j == i) =>
    (((i == j) => ((2 + 2) = 3)) &&
     ((i != j) => ((array[i] + 2) = 3))))
   &&
   ((j != i) => ((1 + 2) = 3))
  )
```

Example C.5 (A loop invariant) Consider following C# code:

```
[XMS_Spec( "x > 0 && y > 0",
  "VALUE = GCD(x,y)",
  "GCD(x,y)=GCD(V_0,V_1):0:V_0,V_1" )]
public int GCD( int x, int y )
{
  int k = x;
  int l = y;

  while ( k - l != 0 )
  {
    if ( k > l )
      k -= l;
    else
      l -= k;
```

```
  }

  return k;
}
```

The C# compiler output for this method is as follows:

```
.method public hidebysig instance int32 GCD(int32 x, int32 y) cil managed
{
    .custom instance
    void [UWr.XMS.Base]UWr.XMS.Base.XMS_Spec::.ctor(string, string, string) =
    { string('x > 0 && y > 0')
      string('VALUE = GCD(x,y)')
      string('GCD(x,y)=GCD(V_0,V_1):0:V_0,V_1') }
    .maxstack 2

    .locals init (
        [0] int32 k,
        [1] int32 l)
    L_0000: ldarg.1     // load x
    L_0001: stloc.0     // k = x
    L_0002: ldarg.2     // load y
    L_0003: stloc.1     // l = y
    L_0004: br.s L_0014 // jump to the loop condition check
    L_0006: ldloc.0     // load k
    L_0007: ldloc.1     // load l
    L_0008: ble.s L_0010 // jump if k <= l
    L_000a: ldloc.0      // load k
    L_000b: ldloc.1      // load l
    L_000c: sub          // k - l
    L_000d: stloc.0      // k = k - l
    L_000e: br.s L_0014  // jump to the loop condition check
    L_0010: ldloc.1      // load l
    L_0011: ldloc.0      // load k
    L_0012: sub          // l - k
    L_0013: stloc.1      // l = l - k
    L_0014: ldloc.0     // load k
    L_0015: ldloc.1     // load l
    L_0016: sub         // k - l
    L_0017: brtrue.s L_0006 // jump if k - l != 0
    L_0019: ldloc.0
    L_001a: ret             // return k
}
```

The VCGen produces following predicate for above code:

```
forall x. forall y.
((x > 0) && (y > 0)) =>
 ((((x - y) = 0) => (x = (GCD(x , y)))) &&
  (((x - y) != 0) => ((GCD(x , y)) = ((GCD(x , y)) &
```

```
(forall V_0__1. forall V_1__1.
  ((GCD(x , y)) = (GCD(V_0__1 , V_1__1))) =>
    (((V_0__1 > V_1__1) =>
      ((((((V_0__1 - V_1__1) - V_1__1) - V_1__1) = 0) =>
         (((V_0__1 - V_1__1) - V_1__1) = (GCD(x , y)))) &&
       (((((V_0__1 - V_1__1) - V_1__1) - V_1__1) != 0) =>
          ((GCD(x , y)) = (GCD((V_0__1 - V_1__1) , V_1__1)))))) &&
     ((V_0__1 <= V_1__1) =>
      ((((V_0__1 - ((V_1__1 - V_0__1) - V_0__1)) = 0) =>
         (V_0__1 = (GCD(x , y)))) &&
       (((V_0__1 - ((V_1__1 - V_0__1) - V_0__1)) != 0) =>
          ((GCD(x , y)) = (GCD(V_0__1 , (V_1__1 - V_0__1)))))))))))))
```

Index

Class Invariants, 40
Common Certificate Specification, 69
Common Language Infrastructure, 17
Common Language Runtime, 17
Common Language Specification, 17

Design by Contract, 11

Garbage Collector, 32

IL
    execution, 19
    fields, 25
    inheritance, 25
    instruction set, 26
    Intermediate Language, 17
    just-in-time (JIT) compilation, 19
    language cooperatibility, 17
    metadata, 18
    methods, 25
    types, 24

Language Based Security, 7

Microsoft Intermediate Language, 17

Proof Carrying Code, 12
    general theorem, 14
    protocol, 13

safety policy, 7
    compositionality, 10
    for Design by Contract, 40
    modularization, 10
Symbolic Evaluator
    aka SE, 53
    for Design by Contract, 54

Verification Condition, 12
    generator, VCGen, 14

XMS
    method specification, 40