JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 2
Special issue: .NET: The Programmer's Perspective: ECOOP Workshop 2003

CIL + Metadata > Executable Program

Giuseppe Attardi, Antonio Cisternino and Diego Colombo, University of Pisa, Italy

Abstract

Execution environments like Java Virtual Machine and Microsoft CLR rely on executables containing information about types and their structure. Method bodies are expressed in an intermediate language rather than machine dependent code to allow verification. Although intermediate language and metadata are required by the execution engine, the information available can be used for other purposes. In this paper we present an application that relies on the rich binary format of CLR to translate the intermediate language code into Lego Mindstorms bytecode. All programming languages targeting the Common Language Infrastructure can be used to program the programmable Lego brick. We exploit the information available to automatically distribute a program between a robot with very limited abilities and a standard PC.

1 INTRODUCTION

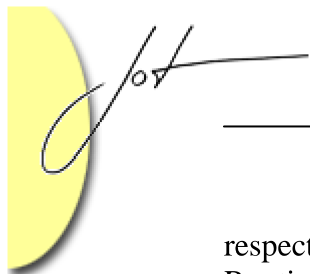
Intermediate languages have been used since the early days of computer science. A common approach to implement efficient interpreters has been employing p-code based systems to reduce parsing costs at runtime (see the introduction of [Krall98]).

In the last years the number of languages based on virtual machines is substantially increased. There are several reasons supporting this trend: hardware is becoming ever faster and we can pay some overhead to get more reusability, security and robustness from our programs; programming has become a hard task requiring an ever increasing number of services.

Garbage collection, libraries of precooked functionalities are often considered a requisite for a programming language. Programming languages based on virtual machines allow programs to be run across different platforms at the only cost of porting the execution environment rather than having to recompile every program. Virtual machines also offer the opportunity of achieving better security: the execution engine mediates all accesses to resources made by programs verifying that the system can't be compromised running applications.

Among the crowd of execution environments two notable examples emerges because of their large use in real world applications and a structure that is more complex with

Cite this article as follows: Guiseppe Attardi et al.: "CIL + Metadata > Executable Program", in *Journal of Object Technology*, vol. 3, no. 2, Special issue: .NET: The Programmer's Perspective: ECOOP Workshop 2003, pp. 19-26. http://www.jot.fm/issues/issue_2004_02/article2



respect to the others: Java Virtual Machine [LiYe99] and Microsoft Common Language Runtime (CLR) [EC335, ISO271]. Both environments distinguish somewhat themselves from the others because of the set of services that are provided at runtime. In particular types and verification have a significant impact on the overall design of the runtime. A major consequence of this fact is that the amount of information about programming abstractions is not thrown away during compilation in favor of a simpler, though equivalent, program expressed in a bytecode with simple instructions.

In this paper we discuss how the metadata and the intermediate language used to describe types of such runtimes can be useful for purposes other than execution. Tasks such as static analysis or bytecode manipulation are simplified because of the abstraction level provided by the intermediate language. To make our statement more concrete we describe a compiler that translates a particular class of .NET binaries into programs executable on the Lego's programmable brick RCX [HaLP99].

2 WHAT KIND OF INFORMATION IS AVAILABLE?

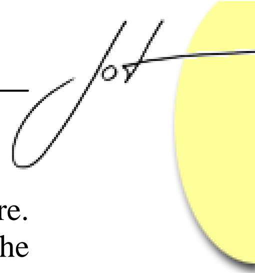
Describing a type-system is a tough job. In Partition II of Common Language Infrastructure (CLI) standard [EC335, ISO271] the metadata format used by .NET assemblies is defined. A single assembly describes a set of types and may contain references to other assemblies. Thirty-six tables are used to describe all the types contained in the assembly and their relation with types contained in other assemblies.

Metadata contain all the information related to types and their structure. Both in JVM and CLR binary formats it is possible to include additional information to metadata[1]. Method bodies are defined using the intermediate language, though they are not accessible through reflection abilities. Nevertheless several libraries have been developed to address this issue; in our system based on .NET we used CLIFileReader library [CistCFR].

The choice of representing types and code in intermediate language form, rather than machine code, is somewhat constrained because of design goals. Without information on types it's almost impossible to have general support for dynamic loading of modules (one weakness of COM [Rog97] was the incomplete type system), reducing reuse of software. Besides the ability of verifying that types are used correctly help to avoid memory corruption due to misuse of programming abstractions: this contributes to reduce the corruption of the execution state and implement security checks.

Types are good for software reuse because are one of the foundations of modern programming languages. Traditionally runtimes, like C runtime or even ML [OCVM] runtime, share a little amount of programming abstractions with the programming

[1] In .NET the ability of annotating metadata is exposed in programming languages such C# [EC334, ISO270]: all programming elements exposed through reflection (types, methods, assemblies and so on) can be annotated by means custom attributes. Java bytecode [LiYe99] can also contain custom information (class file attributes) though this feature isn't exposed to the language: it was conceived to support programming tools like debuggers.



language: in C just numbers are the same used by the processor, in ML a little bit more. When types become a shared abstraction between the execution environment and the programming language a larger amount of information is made available about a program to the runtime and to all the other programs interested in code analysis. Partial evaluators and programs alike can even execute these binaries with different semantics from the one of the execution environment. The simplicity of manipulate intermediate language and metadata makes possible code analysis that would be hard to do in other contexts. In [MaYo01, TSNP02, Cist03] are reported examples of analysis and manipulation of binaries for CLR and JVM.

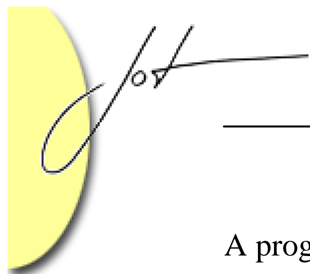
Besides ordinary code analysis, the fact that types are shared between execution environment and the programming language implies that it is possible to provide libraries without implementation. The programmer makes use of such libraries and its invocation to library methods and types are used as placeholders into the binary format for further processing. After compilation programs may manipulate the output looking for special patterns inside the intermediate language, types and metadata. The post processing may be done for several reasons: in [Cist03] it is done for runtime code generation; a post processor would optimize patterns deriving from the use of domain specific operators; some aspect could be interwoven into the code; the executable is translated into an executable for a different platform.

In the rest of this paper we outline the structure of a compiler we developed to translate .NET assemblies into programs for Lego Mindstorms. The compiler is an example of how the information contained into binaries can be used for purposes different from execution.

3 PROGRAMMING LEGO BRICK IN C#

The following program is a simple example:

```
public class SimpleBrick : RCX2 {
  public int guard;
  [FunctionType(Function.Task, 0)]
  public void Main() {
    guard = 1;
    while (guard != 0) {
      PlaySystemSound(Sounds.sweepdown);
      Wait(50);
    }
    }
  [FunctionType(Function.Task, 1)]
  public void Alert(){
    while (guard != 0)
      guard = Sensor1();
  }
}
```



A program that should be executed on the RCX brick is contained in a single class; in our example `SimpleBrick`. We note that the class inherits from a base class representing the type of the brick[2] that we want to program, in the example `RCX2`.

The class has a single field called guard that is used in the two methods of the class `Main` and `Alert`. Lego VM supports up to ten threads executing tasks concurrently. We have used custom attributes to define the mapping between class methods and brick tasks. The custom attribute `FunctionType` is a trivial object whose purpose is to indicate to the compiler how to deal with it. In the example the two methods are mapped into brick tasks.

It is worth noting that we could have used the `Thread` class to represent tasks within the brick. We have avoided this solution because a brick tasks are different from threads, making hard to map the abstractions provided by .NET threads into the simpler task. Moreover from a teaching standpoint it is better to think that methods are executed by different threads rather than having to declare how to spawn threads. Thus we have decided to not expose any threading facility and provide a simpler interface.

The sample program activates two tasks on the brick: the task Main polls the field guard until it becomes 0, at each poll it plays a beep and waits 50 milliseconds before the subsequent check. The task associated to method `Alert` simply reads the value from a sensor and stores it into guard; also in this case when the value read from the sensor number 1 is 0 the task exits.

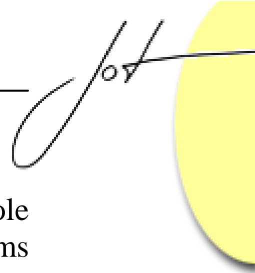
It is important to note that methods `Sensor1`, `Wait`, and `PlaySystemSound` are methods inherited from `RCX2` class. We rely on the type checking to constrain the set of available functions into the class. Type checking is also used to check constants like `Sounds.Sweepdown`.

So can we use windows or databases on the small brick? Of course not! The compiler is aware of the target brick and it checks that the intermediate language associated with a method is compatible with the expressive power of the virtual machine running on the brick.

We have shown a C# program, but we could have chosen also other languages: Visual Basic, Java Script, SML and all the other languages available for .NET. This is possible because we translate the output of the compiler: in few months we have more than doubled the number of languages available to program Lego Mindstorms.

Since its launch many people have contributed in developing programming languages and tools to control Lego Robots. At university of Berlin gcc has been modified to generate code for the processor embedded into the brick. A popular programming language for Lego Mindstorms is Not Quite C (NQC) [NQC] which is a language derived from C. A significant effort has been spent in implementing a full programming language that doesn't provide even the `for` command. An attempt of implementing a Java Virtual Machine for the Lego Brick is ongoing as an open source initiative.

[2] There are different versions of the programmable brick: RCX 1.0, RCX 2.0, Scout, SpyBot and CyberMaster.



We believe that our approach has the advantage of exploiting the whole infrastructure provided by .NET providing the same functionalities of the other systems obtained with a small amount of effort.

4 THE COMPILER

The compiler takes as input a .NET assembly. The first step of compilation inspects the assembly using reflection facilities provided by CLR looking for classes that inherits from the classes representing known bricks.

When a class that should be compiled is found, the compiler inspects its methods and fields. Only methods labeled with custom attributes are considered and the compilation fails is class fields are of types different than `int`. This limitation is because the Lego VM [LegoSdk] supports only integer types. Compiled methods should have the signature void `f()`.

The compilation of method bodies requires a mapping from CIL op-codes into Lego VM ones. This conversion is not straightforward because the Lego VM is register based whereas CLR is a stack based VM.

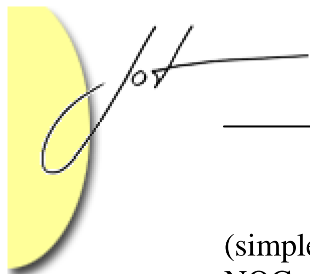
The Lego brick provides 32 global integer variables (available from all tasks) and 16 integer variables local to each task. We mapped class fields into global variables; local variables have been used to emulate the operand stack and (integer) variables local to methods. Of course the maximum stack height (available in the metadata) together with the number of local variables used in a method shouldn't be greater than 16, otherwise the compiler raises an error.

Lego VM provides special operators to read sensors and control motors. We use instance method inherited from the base class to represent these operators. These methods have a dummy implementation and are used only to indicate where special instructions should be generated.

The instance method invocation has a fingerprint easy to detect inside intermediate language. The first step consists in loading the `this` reference on the operands stack with the instruction `ldarg.0`. Then a sequence of instructions follows to load the arguments that should be passed to the method. Finally the callvirt instruction is used to invoke the method. The compiler recognizes such patterns and replaces these instructions with the appropriate Lego instruction.

This approach can be used also to provide advanced functionalities such as automatic distribution of computations between the brick and the PC. When the compiler finds a method invocation that doesn't corresponds to a special method it generates an RPC invocation that sends the method request through the onboard IR port and waits for an answer from the PC. The server on the PC can also be automatically generated exploiting metadata.

In addition to IL patterns recognized to generate special operators, the compiler looks for patterns that can be optimized generating smaller code. The result of this



(simple) optimization step is that the generated code has the same size of the output of NQC compiler for the same program.

5 CONCLUSIONS

Development of Mindstorms compiler taught us that the information available into executables files for JVM and CLR are so rich that these files can be used for purposes different from simple execution. Moreover sharing of the same notion of type between programming language and execution environment allow the development of libraries that provide the illusion of doing something to the programmer. In fact these libraries are used to encode information into executables that can be used by other programs, together with metadata, to manipulate the binary code.

We used this technique to compile a particular subset of .NET classes into programs that are executed on the Lego Mindstorms. The compiler has been developed in few months as a toy project; this fact shows how powerful can be the manipulation of executables: we had the opportunity of focusing our effort on the translation phase rather than spending a large amount of time in implementing our own programming language.

Although we have used .NET the same approach could have been employed for Java bytecode, though .NET assemblies contain more information than Java's class files.

REFERENCES

[CTHL] Calcagno, C., Taha, W., Huang, L., Leroy, X., *A Bytecode-Compiled, Type-safe*, Multi-Stage Language, http://citeseer.nj.nec.com/460583.html.

[Cist03] Cisternino, A., "Multi-Stage and Meta-Programming Support in Strongly Typed Execution Engines", PhD Thesis, May 2003, available at http://www.di.unipi.it/phd/tesi/tesi_2003/PhDthesis_Cisternino.ps.gz.

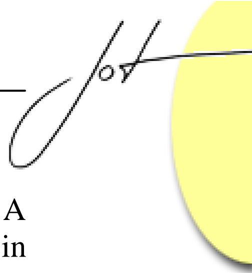
[EC334] ECMA 334, "C# Language Specification", http://www.ecma.ch/ecma1/STAND/ecma-334.htm.

[EC335] ECMA 335, "Common Language Infrastructure (CLI)", http://www.ecma.ch/ecma1/STAND/ecma-335.htm.

[ISO270] ISO/IEC 23270, "Information technology - C# Language Specification", available at http://www.iso.org/

[ISO271] ISO/IEC 23271, "Information technology - Common Language Infrastructure", available at http://www.iso.org/

[KCC99] Kamin, S., Callahan, M., Clausen, L., "Lightweight and generative components II: Binary-level components", in *Proceedings of SAIG00*, 28-50, 1999.



[MaYo01] Masuhara, H., and Yonezawa, A., "Run-time Bytecode Specialization: A Portable Approach to Generating Optimized Specialized Code", in *Proceedings of Programs as Data Objects, Second Symposium,* PADO 2001.

[TSNP02] Tanter, E., Ségura-Devillechaise, M., Noyé, J., Piquer, J., "Altering Java Semantics via Bytecode Manipulation", in *Proceedings of Generative Programming and Component Engineering (GPCE)*, LNCS 2487, 283-298, 2002.

[Krall98] Krall, A., "Efficient JavaVM Just-in-Time Compilation", *International Conference on Parallel Architectures and Compilation Techniques*, ed. Jean-Luc Gaudiot, North-Holland, Paris, 1998, pp. 205-212.

[CistCFR] Cisternino, A., "CLIFileReader library", http://dotnet.di.unipi.it/MultipleContentView.aspx?code=103.

[LiYe99] Lindholm, T., and Yellin, F., *The Java™ Virtual Machine Specification*, Second Edition, Addison-Wesley, 1999.

[HaLP99] Hautop, H., Lund, and Pagliarini, L., "Robot Soccer with LEGO Mindstorms", *Lecture Notes in Computer Science* 1604, 1999, http://mindstorms.lego.com/eng/community/resources/default.asp.

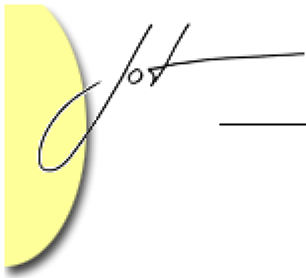
[OCVM] OCaml VM, http://pauillac.inria.fr/~lebotlan/docaml_html/english/.

[Rog97] Rogerson, D., *Inside COM*, Microsoft Press, Redmond, Wa, 1997.

[NQC] "NQC" web site, http://www.baumfamily.org/nqc/.

[TinyVM] "TinyVM" web site, http://tinyvm.sourceforge.net/.

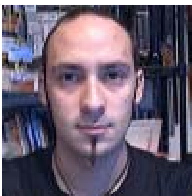
[LegoSdk] Lego Mindstorms SDK, http://mindstorms.lego.com/eng/community/resources/default.



About the authors

Giuseppe Attardi is professor of Computer Science at the Dipartimento di Informatica of Università di Pisa, where he currently teaches Advanced Programming. His research interests are meta-programming and reflection, search engines and question answering. He can be reached at attardi@di.unipi.it.

Antonio Cisternino is research fellow at the Dipartimento di Informatica of Università di Pisa. His current research is on runtime code generation and multi-stage programming on execution environments like JVM and CLI. He can be reached at cisterni@di.unipi.it.



Diego Colombo is student of Computer Science at Università di Pisa. He has a three years degree. He is interested in robotics, computer vision, 3D graphics. He can be reached at colombod@di.unipi.it.