A Process Language Runtime for the .NET Platform

Einar Egilsson

Technical University of Denmark
Kongens Lyngby 2009

Abstract

Process languages, also known as process algebras or process calculi, are languages that are built up of distinct processes communicating with each other. Different process languages have different constructs, but most of the prominent ones have a common subset of constructs. That subset includes action prefixing, parallel composition of subprocesses and non-deterministic choice between paths.

The .NET platform is a popular development platform. One of its strengths is that it supports multiple languages running on the same underlying virtual machine. The languages compile down to a common bytecode format which means that the languages can interoperate and different parts of the same application can be built in different languages.

This thesis explores how well process languages can be integrated into the .NET environment and how they can interoperate with code written in other languages. The design and implementation of an extensible compiler back-end and a runtime library for process languages are presented, as well as two case studies of languages implemented using the common compiler and runtime, and a graphical tool to interact with running process language applications. Finally, a quick overview is given of how to integrate a process language into a state-of-the-art integrated development environment.

Preface

This report is my Master of Science Thesis. It was written between January 15th 2009 and July 15th 2009, in the Language-Based Technology Group of the Computer Science and Engineering section at the Informatics and Mathematical Modeling department of the Technical University of Denmark. My supervisor for this work was Associate Professor Christian W. Probst.

I would like to thank the following people:

My supervisor Christian W. Probst who supported my initial idea for the thesis, and was always available to answer questions and provide support.

Henrik Pilegaard and Sebastian Nanz, whose course, *Process Modelling and Validation*, initially got me interested in process algebra.

Flemming and Hanne Riis Nielson and all the other members of the Language-Based Technology group who helped me realize where my interests lay within Computer Science.

Most of all I thank my wife Karen, who moved with me to another country so that I could study at DTU and always supported me during my Masters studies. *Thank you.*

Lyngby, July 2009

Einar Egilsson

Contents

Abstract **i**

Preface **iii**

1 Introduction **1**
1.1 Thesis Objectives . . . 2
1.2 Thesis Outline . . . 3

2 Background **5**
2.1 Process Languages . . . 5
2.2 The .NET Framework . . . 10
2.3 Summary . . . 14

3 Process Language Runtime **15**
3.1 Inspiration . . . 15
3.2 Overview . . . 16
3.3 Abstract Syntax Tree . . . 16
3.4 Runtime Library . . . 27
3.5 CIL Structure of a Process Language Application . . . 30
3.6 Summary . . . 38

4 Analysis and Optimization **39**
4.1 Analysing process algebra . . . 39
4.2 Analyses . . . 41
4.3 Summary . . . 49

5 CCS Implementation **51**
5.1 Overview . . . 51
5.2 Syntax and semantics . . . 52

5.3 Value passing CCS 53
5.4 Implementation 56
5.5 CCS example system 59
5.6 Summary 63

6 KLAIM Implementation 65
6.1 Overview 65
6.2 Syntax and semantics 67
6.3 Implementation 70
6.4 KLAIM invoice system example 78
6.5 Summary 82

7 Interactive Process Viewer 83
7.1 Overview 83
7.2 Architecture 84
7.3 Summary 87

8 Integrated Development Environment 89
8.1 Choice of Integrated Development Environment 89
8.2 Building a language service 91
8.3 Summary 101

9 Final Considerations 103
9.1 Related work 104
9.2 Further work 106
9.3 Conclusions 110

A Software 113
A.1 Licensing and availability 113
A.2 Source code 114
A.3 CCS Compiler 115
A.4 CCS Visual Studio Integration Package 116
A.5 KLAIM Compiler 117
A.6 Process Viewer 118

B Generated bytecode 121
B.1 Header 122
B.2 Main method 122
B.3 StartProc 123
B.4 ActionPrefix 124
B.5 ValuePassSend and ValuePassReceive 125
B.6 NonDeterministicChoice 128
B.7 ParallelComposition 131
B.8 MethodCall 135
B.9 Restrict 136

B.10 Relabel 140

Chapter 1

Introduction

In the last decade or so, programming languages have increasingly started to target virtual machines instead of specific physical machine architectures. This approach has a number of benefits. Many virtual machines have implementations on different machine architectures and operating systems, which enables an application developer to write applications in a language that targets the virtual machine and getting the benefit of his application running on multiple architectures and operating systems for free. Another benefit of virtual machines is that code written in different languages can interoperate, allowing developers to write each part of their application in the language best suited for the job. The two most prominent virtual machines used today are the Java Virtual Machine (JVM) and the Common Language Runtime (CLR). The JVM was originally created by Sun Microtechnologies but several implementations are now available by many vendors. The CLR was created by Microsoft for the Windows operating system, but an open source version, Mono, which works on Unix and Linux operating systems is also available, which means that .NET can be used to build *cross-platform* applications. Both the JVM and the CLR also have a number of different programming languages that target them, so developers are not tied to a particular programming language if they want to develop for the virtual machines.

Process languages are a class of languages that are made up of distinct processes communicating with each other. Some examples of process languages are *Cal-*

culus of Communicating Systems (CCS), *Communicating Sequential Processes* (CSP), π *calculus* and *Kernel Language for Agents Interaction and Mobility* (KLAIM). The process languages mentioned here have a number of traits in common, including action prefixing, non-deterministic choice and parallel composition. This leads to the assumption that maybe some of these traits can be abstracted away into a common framework, so that each implementation of a process language does not have to implement them separately.

Implementing a process language and having it target a popular virtual machine such as the CLR is an interesting proposition. A number of issues can be explored. How well does the instruction set of the virtual machine fit the execution model of the process language? Is there benefit in having access to the large standard library that comes with the CLR? Is it feasible to write parts of a process language application in another language that targets the virtual machine, for instance numerical functions? A number of advanced tools and integrated development environments exist for CLR and JVM languages, can process languages make use of them?

1.1 Thesis Objectives

This thesis presents the design and implementation of the *Process Language Runtime*, hereafter referred to as the PLR. The PLR consists of two main components. First, it contains an extensible abstract syntax tree, which models common idioms of process languages, and can compile itself to .NET bytecode. Secondly, it contains a runtime library that is used by process applications that have been compiled from the PLR syntax tree.

Integration with the .NET platform is also explored, e.g. how to allow process language applications to call code developed in other .NET languages and whether it is possible to use existing .NET development tools to aid in writing process language applications.

A tool to interact with running process applications is also developed, this tool has a graphical user interface and is meant to give more insight into what is happening when a process language application is running and allow users to select which paths are taken during execution.

Finally, the thesis presents two case studies of process language implementations that were made using the PLR. The languages implemented were CCS (Calculus of Communicating Systems) and a subset of KLAIM (Kernel Language for Agents Interaction and Mobility).

The work carried out consists of the following parts:

- Design and implementation of the PLR abstract syntax tree and compiler, including static analysis and compiler optimizations.
- Design and implementation of the runtime library.
- Implementation of the process language CCS.
- Implementation of the process language KLAIM.
- Design and implementation of a tool to interact with running process language applications.
- Integration of the CCS language into Visual Studio, a state-of-the-art integrated development environment for .NET development.

1.2 Thesis Outline

The thesis consists of nine chapters, of which this introduction is the first. The rest are as follows:

Chapter 2 gives some background on process languages in general and their common properties, and some technical background on the .NET platform. The concepts presented there are useful for understanding the architecture of the PLR.
Chapter 3 describes the Process Language Runtime, both the syntax tree and runtime library and shows how they are designed and implemented.
Chapter 4 describes the static analysis and optimizations that are performed before compilation.
Chapter 5 is a case study of an implementation of the CCS process language.
Chapter 6 is another case study, this time of an implementation of the KLAIM process language.
Chapter 7 is about the graphical tool Process Viewer and how it can be used to interact with process language applications.
Chapter 8 gives an overview of how the CCS language was integrated into Visual Studio 2008.
Finally, Chapter 9 contains concluding remarks and an exploration of related work. Ideas for further development of the PLR are also discussed, including how some other common process languages could potentially be implemented using the PLR.

In addition, there are two appendices:

Appendix A is about the practical aspects of the software developed during the course of the project, where it can be downloaded, how it is licensed and how it can be configured and run.
Appendix B shows a small CCS system and all the generated bytecode for it.

Chapter 2

Background

To fully understand the issues involved in creating the Process Language Runtime a little background knowledge is required. First, process languages are explained, their history, common properties and practical applications. Secondly, the .NET framework is presented and its technology explained.

2.1 Process Languages

2.1.1 Overview and history

Process languages, also known as process algebras or process calculi, are a family of languages to formally model concurrent systems. These languages describe the systems at a high level of abstraction, as interactions, communications, and synchronizations between a collection of independent processes. This is typically done using only a handful of constructs, many of which are shared between different process languages, albeit with different concrete syntax. These common constructs are explained further in Section 2.1.2. Some of the more prominent process languages today include *Calculus of Communicating Systems* (CCS), *Communicating Sequential Processes* (CSP), *π-calculus* and *Kernel Language for Agents Interaction and Mobility* (KLAIM).

Algebraic laws have been defined for these languages that allow process descriptions or equations to be analyzed and manipulated, and permit formal reasoning about equivalences between different processes, using for instance bi-simulation. In this paper we do not focus on these algebraic properties but instead concern ourselves with implementations of process algebras as programming languages. For those interested, a good explanation of the algebra involved in one such language, CCS, can be found in [1].

Two of the most important figures in the history of process algebras are Robert Milner and C.A.R. Hoare. Milner published a number of papers [19, 20, 21] throughout the 1970's about concurrency and possible formal semantics for analyzing concurrent systems. In 1980 he published [22] which introduced *Calculus of Communicating Systems*. He continued working on concurrent systems and in 1989 published [24] which introduced *π-calculus*, a successor to CCS. Hoare started working on process algebras at a similar time and in 1978 published [16] where he introduced *Communicating Sequential Processes* or CSP. These two algebras, CCS and CSP, have become the basis on which much of the later work in this field derives from. Both Milner and Hoare have continued working with concurrent systems and have published a number of papers refining and further extending CCS and CSP.

Practical applications of process algebra are many. They have been used to model real world systems, verify absence of deadlocks and break cryptographic protocols to name a few. They have also influenced a number of mainstream programming languages. One example is the Erlang programming language. It was developed by the Ericsson telecommunications company and its main strength is concurrency. It has the notion of processes that communicate through message passing, and the core language has been modelled in π-calculus [28]. Other examples are the occam programming language which builds on CSP, and occam-pi [30] which incorporates ideas from both CSP and π-calculus.

2.1.2 Common constructs

The syntax for the common constructs varies between different process languages, in this section we shall use the syntax from Calculus of Communicating Systems (CCS) to demonstrate the concepts behind the constructs.

Parallel composition is the key construct which separates process algebra from sequential modes of computation. With parallel composition, two or more processes can run independently of each other at the same time. Parallel composition is typically represented with the | character, so for two parallel processes, P and Q, we write $P \mid Q$ to indicate that they run in parallel.

A nil process is a process that does nothing and cannot interact with any other processes. It has different representations in different algebras, common symbols for it include **nil**, **0** and **STOP**. The purpose of the nil process is to be an anchor upon which more interesting processes can be generated. An usual pattern is for a process is to first perform one or more actions and then turn into the nil process, which signifies that the process has run its course.

Message passing through channels is the way processes interact with each other. One process sends an outbound message on a particular named channel and another process accepts a message on the same named channel.

Example 2.1

$$\begin{aligned} \mathrm{P} &\overset{\text{def}}{=} coffee \mathbin{.} 0 \\ \mathrm{Q} &\overset{\text{def}}{=} \overline{coffee} \mathbin{.} 0 \end{aligned}$$

In Example 2.1 above the process P listens on the coffee channel while process Q sends on it. A process that is sending or receiving on a particular channel is blocked until another process performs the opposite operation on the channel. A synchronization happens between one sender and one receiver, if two processes had been ready to receive on the coffee channel at the same time then one of them would be chosen and the other would remain blocked. Channels are often given descriptive names to indicate their purpose, we read the example above as *P receives coffee from Q*. However, values can also be passed along channels, and can then be bound to variables in the receiving process. An example of this is shown in Example 2.2 below.

Example 2.2

$$\begin{aligned} \text{Teacher} &\overset{\text{def}}{=} \overline{grade(12)} \mathbin{.} 0 \\ \text{Student} &\overset{\text{def}}{=} grade(x) \mathbin{.} 0 \end{aligned}$$

In this example the Teacher process sends the value 12 on the grade channel. The Student process receives the message on the channel and binds the value to a variable x which can then be used in the continuation of the process.

Action prefixing is how sequential processes are built up. A process is prefixed with an action, meaning that first an action is performed and then the process continues as the prefixed process. The syntax for this is generally a dot between the action and the following process. An example of action prefixing is $a \mathbin{.} P$, here action a is performed and then the process continues as P. P itself could also be an action prefixed process, it is straightforward to see how this can be expanded into a series of actions, e.g. $a \,.\, b \,.\, \overline{c} \,.\, \mathrm{P}$. The final process P could either be the *nil process* or a *process constant*, which is further explained below.

Process constants are labels given to particular processes to identify them. In Example 2.2 we saw two examples of process constants, Teacher and Student. These constants can then be used in process descriptions to indicate that a given process turns into another process. In Example 2.3 we see that a CoffeeMachine process first accepts a coin, then outputs a coffee and then turns back into a CoffeeMachine process. This can be expanded into an endless series of $coin \,.\, \overline{coffee} \,.\, coin \,.\, \overline{coffee} \,.\, coin \,.\, \overline{coffee}$ etc. Recursive process definitions like these are used instead of looping constructs which process languages generally do not have. Of course the CoffeeMachine process could just as well have turned into any other process at the end, it does not have to only turn into itself.

Example 2.3

$$\text{CoffeeMachine} \stackrel{\text{def}}{=} coin \,.\, \overline{coffee} \,.\, \text{CoffeeMachine}$$

Nondeterministic choice is a method for processes to choose between two or more actions that the process can perform. The process is free to choose arbitrarily which action to take. Example 2.4 shows how process P can perform one of actions a, b or c. The choice is not made until at least one of the channels has a corresponding process outputting on it that the choosing process can synchronize with. There is no guarantee that the probability between choices is fair, a recursive process might choose a every time, or might make each choice exactly 33% of the time. An important thing to notice is that different paths in a process language are never joined again. This, coupled with the fact that process languages generally do not have loop structures, means that the control flow graph for process languages is always a tree.

Example 2.4

$$\mathrm{P} \stackrel{\text{def}}{=} a \,.\, 0 + b \,.\, 0 + c \,.\, 0$$

Restriction hides channel events within a process from the outside world. This can be used to simulate a machine that has internal workings which are restricted, and a public interface which external processes can synchronize with. Example 2.5 shows a process P that hides the channel a. That means that the two parallel processes that P is composed of can only synchronize with each other on the a channel, not with an outside process. The *observable* behaviour of P is that it only outputs on the b channel and then terminates.

Example 2.5

$$\mathrm{P} \overset{\text{def}}{=} (a \,.\, 0 \mid \overline{a} \,.\, \overline{b} \,.\, 0)$$

Re-labelling is a way to create general processes and make them more specific by substituting their channel names with other channel names. Consider for example the CoffeeMachine process in Example 2.3. We can see that this is really a specific version of a vending machine. To enable re-use for different types of vending machines we could create a generic VendingMachine process that dispenses items, and create specific vending machines by re-labelling those items to specific products. A re-worked example of a CoffeeMachine is shown in Example 2.6 where item is re-labelled to coffee.

Example 2.6

$$\begin{aligned}\text{VendingMachine} &\overset{\text{def}}{=} coin \,.\, \overline{item} \,.\, \text{VendingMachine}\\ \text{CoffeeMachine} &\overset{\text{def}}{=} \text{VendingMachine}_{[coffee/item]}\end{aligned}$$

One final thing that is important about both re-labelling and restriction is that they keep applying to the process even as it invokes another process. In Example 2.7 we see a process *Proc* that relabels the a channel to b. It eventually turns into the *AnotherProc* process. The re-labelling of a to b **still applies** to *AnotherProc* and any processes it might subsequently turn into. A process turning into another process is essentially the same as substituting the process itself for the process constant name. Following that rule we see that the process *Proc* in Example 2.7 could also be written as Proc $\overset{\text{def}}{=}$ $(a \,.\, (a \,.\, 0))[b/a]$ and there it is obvious why the re-labelling applies to the invoked process.

Example 2.7

$$\begin{aligned}\text{Proc} &\overset{\text{def}}{=} (a \,.\, AnotherProc)[b/a]\\ \text{AnotherProc} &\overset{\text{def}}{=} (a \,.\, 0)\end{aligned}$$

These were the main process language constructs, other less common constructs will not be explained here. Some other constructs are however considered in Section 9.2 about further work.

2.2 The .NET Framework

2.2.1 Overview and history

The .NET Framework is a framework from Microsoft for writing software applications. It consists of a virtual machine that runs programs coded specifically for the framework and a large standard library for application developers to use when writing their applications. In addition, two programming languages are included in the default distribution of the framework, C# and Visual Basic.NET. (A third language, J#, was included in earlier versions but has since been dropped).

The original name for .NET was Next Generation Windows Services (NGWS) and its development started in the late 1990's at Microsoft. In late 2000 the first beta versions of .NET 1.0 were released, and the first official version of the .NET framework, 1.0, was released on February 13th, 2002. As of this writing there have been five major releases of the framework, 1.0, 1.1, 2.0, 3.0 and 3.5. With each new version additional features have been added, but not always in the way you would expect. The first three versions, 1.0, 1.1 and 2.0 all contained new versions of the virtual machine, new versions of the compiler for the standard languages and additional libraries. However, version 3.0 of the framework contained only new libraries but no new compilers and no new version of the virtual machine. Version 3.5 then included new versions of C# and Visual Basic.NET and some additional libraries, but again no change to the virtual machine. As a result, the version numbers of the different components of the framework have diverged so when we talk about version 3.5 of the framework, that includes version 3.5 of the libraries, version 2.0 of the virtual machine and version 3.0 of the C# language.

2.2.2 Common Language Infrastructure

The Common Language Infrastructure (CLI) is an open specification developed by Microsoft that describes an executable code and runtime environment. This is Microsoft's specification of the .NET framework, but it has been published under the ECMA-335 and ISO/IEC 23271 standars and so anyone is free to write their own version that follows this specification. Two main alternate versions exist, Mono and DotGNU. Both of these are released under open source licenses and work on multiple operating systems, as opposed to Microsoft's .NET which only runs on the Windows family of operating systems. Microsoft has also released a shared source reference implementation of the CLI specification. None of these other implementations fully implement all the class libraries of the original .NET framework, and typically are about one version behind Microsoft's .NET framework.

The five main components described by the CLI specification are as follows:

1. **The Common Type System (CTS):** a set of types and operations that are shared by all CLI-compliant programming languages.
2. **Metadata:** Any CLI language can access code written in any other CLI language. To achieve this, information about program structure is language agnostic.
3. **Common Language Specification (CLS):** A set of base rules to which any language targeting the CLI should conform in order to interoperate with other CLS-compliant languages. The CLS rules define a subset of the Common Type System.
4. **Virtual Execution System (VES):** The VES is the component that loads and executes CLI-compatible programs.
5. **Common Intermediate Language (CIL):** An intermediate language that is abstracted away from the platform hardware. Upon execution, the platform-specific VES will use a Just-in-time (JIT) compiler to compile the CIL to hardware specific assembly language. Common Intermediate Language is often referred to under the names MSIL (Microsoft Intermediate Language) or simply as .NET bytecode.

2.2.3 Languages

Two programming languages are included in the .NET default distribution. Those are C# and Visual Basic.NET. C# derives its syntax from the C family of languages, and in its first version was almost identical to the Java programming language. Later versions have acquired a number of new features such as lambdas, anonymous delegates and generators. In theory all .NET languages are created equal; in practice C# is first among equals, and the entire standard library is for instance written in C#.

Visual Basic.NET derives from the Basic family of languages. It has more verbose syntax than C# and is the continuation of Microsoft's Visual Basic 6 language. Visual Basic.NET has a number of differences from previous version of Visual Basic though, mainly to fit into the .NET mold. To ease the transition from Visual Basic 6 to Visual Basic.NET, Microsoft included a number of old Visual Basic functions with the .NET framework in the namespace Microsoft.VisualBasic.

In addition to these two main languages, there are dozens of other languages that have implementations targeting .NET. These include well established languages such as C++, Delphi, Lisp, Scheme, Smalltalk, Java (in J#), Python, Cobol, Ruby and JavaScript as well as languages that have been built for .NET from the start, such as F#, Nemerle and Boo.

2.2.4 Virtual machine

The virtual machine, of the .NET framework is named the Common Language Runtime (CLR). It manages the runtime requirements of programs written for .NET and frees the programmer from having to consider specific machine architectures or CPU's, as far as the programmer is concerned the CLR is the (virtual) machine architecture that they are targeting. The CLR also provides other runtime services such as security, exception handling and memory management. Of these, perhaps the most important service provided is memory management, which frees the programmer from allocating and de-allocating memory at runtime. Some of the most common programming errors in languages without memory management, such as C++, have to do with failing to de-allocate memory and thereby causing memory leaks, or accessing memory incorrectly which in turn causes segmentation faults. Programs written for .NET eschew this class of errors completely[1].

The CLR executes programs that have been compiled to the Common Intermediate Language (CIL). The CLR is a stack based virtual machine, which means that it has an evaluation stack where the CIL bytecodes are evaluated. An illustrative snippet of C# code and its corresponding CIL is shown in Figures 2.1 and 2.2. The CIL bytecodes generally fall into four categories:

1. **Load items onto the stack**. These include bytecodes to load integers, floating point numbers, strings, object references, local variables or class fields onto the evaluation stack. These bytecodes take one argument each, the item to be loaded onto the stack.

2. **Store stack items into variables/fields**. These bytecodes take the top

[1]Although .NET programs do not cause memory leaks themselves, they may cause them if they interact directly with other unmanaged code, such as making calls to the operating system functions directly via P/Invoke

item on the stack and store it in a local, global or class member variable. They take one argument, which is a reference to the variable to store in.

3. **Call functions**. These take an argument, that is a reference to the function to call, and then call it with the parameters that are currently on the evaluation stack.

4. **Operate on stack items**. These bytecodes usually do not take any arguments, they simply use values on the stack and push results back onto the stack. Examples of these are bytecodes that add, multiply or divide the top two values on the stack and then push the result back onto the stack.

At the time of execution, the CLR generates native code for the particular machine architecture that it is running on from the CIL bytecodes, this is referred to as Just-in-time compiling, or JIT compiling. A more detailed explanation of that process is outside the scope of this paper, but an overview can be found in [6].

```
using System;
namespace CodeGenDemo {
    class Program {
        static void Main(string[] args) {
            Console.WriteLine("Hello world");
        }
    }
}
```

Figure 2.1: C# Hello World program

```
.class private auto ansi beforefieldinit CodeGenDemo.Program
       extends [mscorlib]System.Object
{
  .method private hidebysig static void  Main(string[] args) cil managed
  {
    .entrypoint
    // Code size       20 (0x14)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr      "Hello world"
    IL_0006:  call       void [mscorlib]System.Console
                         ::WriteLine(string)
    IL_0013:  ret
  } // end of method Program::Main
```

Figure 2.2: C# Hello World program compiled to CIL

2.3 Summary

Having some understanding of process algebra and its constructs and the .NET framework is necessary to understand the architecture of the Process Language Runtime. In this chapter we looked at the basic process algebra constructs that are shared between multiple process algebras and are implemented in the PLR. These are action prefixing, parallel composition, non-deterministic choice, restriction, relabelling and process invocations. The chapter also gave an overview of the underlying technology of the .NET framework, the virtual machine, bytecode format and its associated standards and specifications. In the following chapter we will see how the process algebra constructs explained here are implemented for the .NET virtual machine.

Chapter 3

Process Language Runtime

This chapter presents the design and implementation of the Process Language Runtime, an extensible compiler backend and a runtime library for running process languages on the .NET platform.

3.1 Inspiration

The inspiration for the Process Language Runtime comes from the Dynamic Language Runtime [7] (DLR), a framework from Microsoft for developing dynamic languages on top of the Common Language Runtime. Since the .NET Common Intermediate Language is statically typed it is ill-suited for dynamic languages such as Python, Ruby or JavaScript. To clarify, statically typed languages are languages where the type of a variable is known at compile time and the type of a variable never changes, while in a dynamically typed language a single variable can contain objects of different types at different times during program execution. The idea of having a common abstract syntax tree for different languages comes from the DLR. However, the DLR was not directly used for this project since its abstract syntax tree is mainly concerned with traditional constructs for imperative programming languages and constructs to support dynamic typing, while the purpose of this project is to provide constructs common to process languages, and dynamic typing is not a concern.

3.2 Overview

The PLR is a single .NET assembly, named PLR.dll. This assembly contains the abstract syntax tree and associated helper objects as well as all the classes used at runtime. The PLR does not contain any lexer or parser and generally has no notion of any concrete syntax. Creating a lexer and parser is the responsibility of individual language implementations, the PLR takes over once an input file has been parsed and used to construct a PLR abstract syntax tree.

It is important to note that the PLR does not, nor is it meant to, support all constructs of all process languages. Creating a superset of all existing process languages has never been the goal, instead the goal is to provide a common subset of the most common constructs found in these languages and to make it easy to extend with specific new constructs needed for specific languages. To enable this, the classes and interfaces in the PLR have been engineered to make them easy to subclass and implement.

As the PLR contains classes needed at runtime it must be distributed with any compiled process language application. However, an option is present at the compilation stage that allows the PLR assembly to be embedded in the final compiled executable program, and can optionally embed any additional runtime libraries that specific languages require. This allows a process language program to be distributed as a single file without any external dependencies other than the .NET framework itself.

The PLR is written in the C# programming language using Visual Studio 2008 as the development environment. It has a dependency on the NUnit unit test framework, however this dependency is only needed when running internal unit tests and so does not need to be distributed with the PLR assembly. The source code for the PLR is licensed under the General Public License (GPL) v3.0.

3.3 Abstract Syntax Tree

The PLR abstract syntax tree is the component that generates CIL bytecode to run a process language application. Once an abstract syntax tree has been constructed, a call to a Compile method on the tree's root node with the appropriate parameters will create an executable .NET assembly.

3.3.1 Architecture

The architecture of the abstract syntax tree is based on Object Oriented principles, namely that an object contains data and methods to operate on that data. As such, each node in the abstract syntax tree knows how to compile itself, there is no compiler class, the whole syntax tree is the compiler. Every node in the syntax tree inherits from an abstract Node base class which has an abstract Compile method. Concrete node classes override the Compile method and in it they emit the appropriate byte codes for the language construct that the node represents.

The compilation itself is recursive, calling the Compile method on the root node of the tree will cause it to call the Compile method of its child nodes, who in turn call Compile on their child nodes and thus the compilation propagates throughout the entire tree. The reason for choosing this architecture was to make the syntax tree easily extendable by language implementers, who can add new nodes to represent new constructs.

Concrete nodes typically do not inherit directly from the Node base class, instead they inherit from one of five intermediate classes, Action, Process, Expression, ActionRestrictions or PreProcessActions. Below is a short overview of what each of these classes represents.

Action represents an action taken by the process. This can for instance be sending on a channel, receiving on a channel or calling an arbitrary method. Concrete descendants of this class typically have either no child nodes of their own, or a list of Expression nodes, representing parameters to a method call or values passed through a channel.

Process is the base class for processes. Its descendants include an ActionPrefix class, a NonDeterministicChoice class and a ParallelComposition class. Implementing a new language construct such as replication could be done be creating a new descendant of this class. Child nodes of Process classes vary, the ActionPrefix class for example has one Action child node representing the action about to be performed and one Process child node representing the process that the current process turns into after performing the action. Processes that are a composition of other processes such as ParallelComposition and NonDeterministicChoice have a list of other Process instances as childnodes.

Expression represents an expression such as an arithmetic expression, numeric or string constant, a method call or the value of a variable. Expressions compile in such a way that once they have been evaluated a single value, the result of the expression, is at the top of the evaluation stack. This means that a node that has an expression as a child node can simply call the Compile method on the expression and then emit bytecodes that operate on its result, without caring whether the expression is a huge expression tree or a single constant value. An example of this is the ArithmeticExpression node. In its Compile method it first calls the Compile method of its left child, then its right child and then emits an Add,Sub,Mul or Div bytecode. The child nodes of Expression nodes are invariably Expression nodes themselves.

ActionRestrictions represents a function that restricts actions within a process from synchronizing with other actions outside the process. This is an implementation of the *restriction* process language construct described in Section 2.1.2. It currently has two concrete descendants. One is ChannelRestrictions which restricts channels by name, provided that the names of channels to restrict are known at compile time. The other is CustomRestrictions, that calls a .NET method at runtime for every action and returns `true` it it should be restricted. The method can be written in any language available for the .NET framework, the only requirements are that it takes an IAction object from the PLR runtime library as a parameter and returns a boolean value.

PreProcessActions represents a function that is called for every action that is performed in a process and returns another action. This is used to implement the *re-labelling* process language construct described in Section 2.1.2. Simple re-labelling of channels with names known at compile time is done with a RelabelActions class. It has the names to re-label as child nodes and compiles down to a method that takes in an IAction runtime class and performs simple string substitution on its name. More complicated pre-processing of actions can be achieved with another descendant class, CustomPreprocess. That class compiles down to a method call to a .NET method that takes an IAction as a parameter and returns an IAction as well. This method can be written in any language available for .NET.

Besides all the descendant classes of those five main classes, there are a few classes that inherit directly from the Node class. ProcessSystem is the root node of the entire syntax tree and has a more complicated Compile method than most other nodes, since it takes care of setting up the necessary context for the compilation and creating the actual compiled file, giving it a name and so forth. ProcessDefinition is a simple class that just has a Process child node and a name for the process. Finally, ExpressionList is a convenience class to hold a list of Expression instances.

We now look at a simple example of a coffee machine, CM. The coffee machine accepts a coin as input, then outputs coffee and then makes a non deterministic choice between turning into itself again or turning into a process representing a failure of the coffee machine, CMFAIL. The CMFAIL process accepts a coin and then turns into the nil process without ever returning coffee for the inserted coin.

Example 3.1

$$\mathrm{CM} \overset{\mathrm{def}}{=} coin \,.\, \overline{coffee} \,.\, (\mathrm{CM} + \mathrm{CMFAIL})$$
$$\mathrm{CM} \overset{\mathrm{def}}{=} coin \,.\, 0$$

The syntax tree for the processes in Example 3.1 is shown in Figure 3.1. The names shown in boldface are the names of the node classes, while the text in parentheses shows properties of the nodes. Note that although this particular tree is strictly binary it does not mean that all PLR trees are binary trees. ProcessSystem, ParallelComposition and NonDeterministicChoice nodes can all have 1-n child nodes.

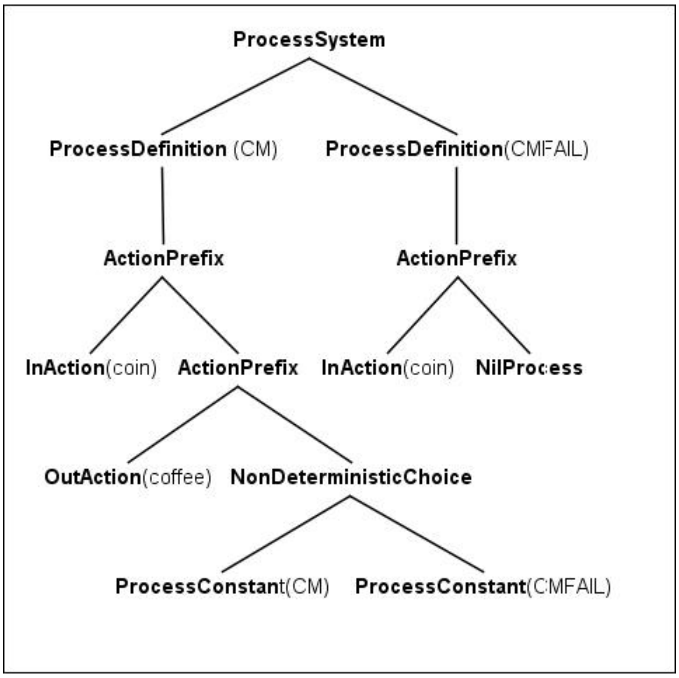


Figure 3.1: PLR abstract syntax tree

```
//Each node in the tree contains this method
public override void Accept(AbstractVisitor visitor) {
  visitor.Visit(this);
}

//The Visit method for ActionPrefix node in a subclass
//of AbstractVisitor.
public override void Visit(ActionPrefix node) {
  //...process the ActionPrefix node here
}
```

Figure 3.2: Examples of Visit and Accept methods

3.3.2 Processing the syntax tree before compilation

There are many reasons why a language implementation might need to process the abstract syntax tree in some way before compilation. An example might be optimization; to fold constant expressions or prune branches of the tree that are sure to never be executed. To support scenarios like this, the PLR makes use of the *Visitor* design pattern. The pattern is a way of separating an algorithm from an object structure upon which it operates. As a result, new operations can be added to existing object structures without modifying those structures. A visitor interface contains one Visit method for each of the classes in the object structure, each class in the object structure then contains an Accept method that takes the visitor interface as a parameter and does nothing except call the visitor's Visit method with itself as a parameter. An example of this is shown in Figure 3.2. This technique, calling the objects Visit methods that immediately calls the visitors Accept methods is known as *double dispatch*. This essentially mimics virtual method overloading, but the added benefit is that the methods can be defined outside the object structure, making it easy to plug in different implementations of the visitor as needed. The visitor class can then contain different implementations of traversing the object structure while calling the Accept method on each of its nodes. A more detailed explanation of the visitor pattern and its benefits can be found in [11, 29].

The PLR contains an AbstractVisitor class which is the base class of all visitor implementations. This was implemented as an abstract class rather than an interface for convenience reasons; the AbstractVisitor provides empty implementations of the Visit method for each of the nodes in the abstract syntax tree, subclasses only need to override the Visit methods for nodes which they are interested in processing. Depth first traversal is a very common method of working with tree structures, to account for that common case the AbstractVisitor contains a VisitRecursive(Node node) method which performs a depth first recursive traversal of the tree, calling each nodes Accept method along the way. A boolean property on the AbstractVisitor named VisitParentBeforeChildren controls whether the parent or child nodes Accept methods are called first during the traversal.

Process algebras are the subject of many academic papers, and it is likely that future users of the PLR might be in the process of writing research papers themselves. For that reason it could be quite useful to be able to get a text representation of the abstract syntax tree, formatted in the LaTeX typesetting format (it certainly has been very useful for this author!). The PLR contains three different formatter classes that generate text representations of the syntax tree in different formats.

- BaseFormatter generates unformatted process algebra text, using the common symbols for constructs like non deterministic choice and parallel composition.
- LaTeXFormatter generates LaTeX source, suitable for copying directly into a LaTeX document.
- HTMLFormatter generates HTML formatted text, suitable for displaying on the web.

The concrete syntax used by all of these formatters is that of CCS, however, since many of the common process algebras use the same symbols for things such as parallel composition and non deterministic choice, formatters for other languages could be implemented simply by inheriting from one of the three aforementioned classes and overriding the formatting methods only for those constructs whose syntax differs from CCS syntax.

The formatter classes are all implemented using the visitor pattern. The benefits of the pattern here are obvious, it is easy to add new formats at a later date without having to alter anything in the syntax tree itself, all code for a particular format is kept in one place and formatter classes do not need to implement tree traversal algorithms themselves. Another example of how formatters can be useful is shown in Section 7.2.3.

Figure 3.3 shows another example of how the visitor pattern can be useful. It is a class that takes binary expressions that contain constants on both the left and right hand side, calculates their results and replaces the ArithmeticBinOpExpression node (which contains two child nodes) with a single Number node containing the result of the expression. The code listing shows the entire ExpressionFolder class, all it needs to do is inherit from AbstractVisitor and override the Visit method that takes ArithmeticBinOpExpression as a parameter. The visitor is executed by calling `folder.VisitRecursive(tree)` where `folder` is an instance of ExpressionFolder and `tree` is any Node in the syntax tree, usually the root. Note that the folding only happens if both the left and right node are Number nodes, however, the fact that the tree is traversed depth first and child nodes are visited before parent nodes ensures that even deeply nested expressions are folded as much as possible. The leafs are visited first and folded if possible, by the time the upper level nodes in the expression tree are visited they will have Number nodes as children where previously were ArithmeticBinOpExpression nodes, and thus they can be replaced as well. (Of course this example is fairly contrived, it is hard to think of a legitimate reason for writing down a large expression where all elements are constants. It does however show how the visitor pattern can be used to implement classes that alter the syntax tree with a minimal amount of code).

```
class ExpressionFolder : AbstractVisitor{

  public override void Visit(ArithmeticBinOpExpression exp) {
    if (exp.Left is Number && exp.Right is Number) {
      int result = 0, leftVal, rightVal;
      leftVal = ((Number)exp.Left).Value;
      rightVal = ((Number)exp.Right).Value;

      if (exp.Op == ArithmeticBinOp.Plus) {
        result = rightVal + leftVal;
      } else if (exp.Op == ArithmeticBinOp.Minus) {
        result = rightVal - leftVal;
      } else if (exp.Op == ArithmeticBinOp.Multiply) {
        result = rightVal * leftVal;
      } else if (exp.Op == ArithmeticBinOp.Divide) {
        result = rightVal / leftVal;
      }
      int pos = exp.Parent.ChildNodes.IndexOf(exp);
      exp.Parent.ChildNodes[pos] = new Number(result);
    }
  }
}
```

Figure 3.3: Expression folder implemented using Visitor pattern

3.3.3 Extensibility

As stated before, one of the goals of the PLR is extensibility, allowing for language implementers to add features and constructs not included in the PLR itself. There are three main methods of extending the PLR. Firstly, language implementers can add new nodes to the abstract syntax tree. These nodes just have to implement the Compile method and then they can be seamlessly integrated with the built-in PLR nodes. Of course it is also possible to inherit from one of the existing nodes and re-use some of the compilation work they do, and simply add extra code before or after the base class's compilation step.

Secondly, the root node of the abstract syntax tree, ProcessSystem exposes the following four events.

1. BeforeCompile occurs before the PLR has performed any compilation. At this point no types or methods exist yet.
2. AfterCompile occurs after the PLR has finished all its compilation but before it creates the executable file. At this point subscribers to this event can access any types or methods created during compilation.
3. MainMethodStart occurs just after the main method of the application has been defined but before any bytecodes have been emitted into it. Subscribers of this event can then inject their own bytecodes at the beginning of the main method if they wish.
4. MainMethodEnd occurs after all bytecodes of the main method have been emitted, except for the final Ret instruction. Again, subscribers of this event can inject their own bytecodes at this point.

All these events have the same signature, they require a CompileEventHandler delegate, which takes a CompileContext object as a parameter. The event subscribers then use the compile context to create types, methods and emit bytecodes at different points in the compilation process.

Finally, the third way to extend the PLR is to write supporting code in another .NET language, writing a seperate runtime library. The KLAIM implementation described in Chapter 6 takes this approach. When a language implementation requires large amounts of supporting code it is inconvenient and error prone to generate all that code by emitting CIL bytecode at compilation time. By creating a runtime library instead, language implementers can get the benefit of programming languages and tools such as C# and Visual Studio when writing the common, re-usable parts of their languages. Then, at compilation time, they can simply emit bytecodes to call code in the runtime library. This is also the approach taken by the PLR itself, which has runtime classes written in C# and emits bytecodes during compilation that interact with these classes.

3.3.4 Code generation

To generate a valid .NET assembly the PLR uses a set of classes that are a part of the .NET framework Base Class Library (BCL). These classes are located in the System.Reflection.Emit namespace. Before explaining more about these classes it is worth going over how a .NET assembly is structured. A .NET *assembly* is an executable file (.exe) or a dynamic link library (.dll). The assembly contains one more *modules*, typically just one. Each module contains one or more *types* (or classes). Types have *fields*, *constructors* and *methods*. At the lowest level, constructors and methods contain CIL bytecodes. Figure 3.4 shows the structure.

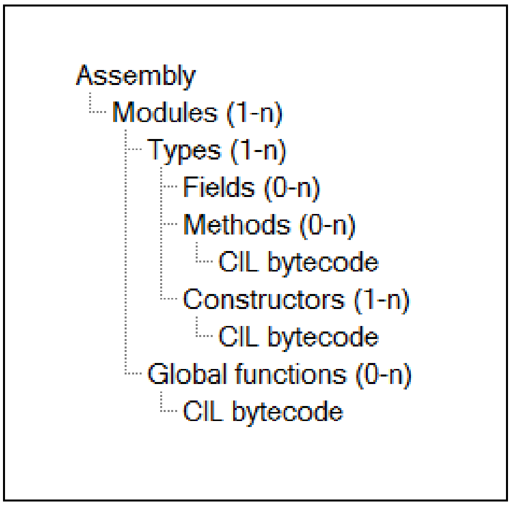


Figure 3.4: The structure of a CIL assembly

The classes in the System.Reflection.Emit namespace match the structure of an assembly. There is an AssemblyBuilder, ModuleBuilder, TypeBuilder, FieldBuilder, ConstructorBuilder and a MethodBuilder. These are instantiated by giving them names and other properties as parameters. The ConstructorBuilder and MethodBuilder have a GetILGenerator method that returns an object of type ILGenerator. That object has direct access to the bytecode stream of the method being created, and contains various overloads of an Emit method that emits bytecodes and their associated arguments. An OpCodes class contains constants for all possible bytecodes that can be emitted by the ILGenerator.

The nodes of the syntax tree gain access to these classes through a CompileContext class which is part of the PLR, and is a parameter to the Compile method implemented by all nodes. The CompileContext class has a number of useful properties that the nodes can access. It exposes the TypeBuilder object of the type currently being built, the ILGenerator object of the method or constructor being built and a symbol table for variables currently in scope. The node can then emit its bytecodes, create new variables or otherwise alter the CompileContext before passing it on to its child nodes Compile methods. Essentially this is a form of distributed compiling, no one node has a complete picture of what is being compiled, each node only has enough information to add its own code to the correct type or method.

3.3.5 Debugging support

One of the benefits of targeting a common virtual machine such as the CLR is is that both a free command line debugger and a free graphical debugger exist that can be used for any programming language that compiles down to the Common Intermediate Language format. The System.Reflection.Emit API offers functionality to emit the necessary debugging symbols to be able to use these debuggers. Emitting debug symbols consists of the following five steps:

1. When the ModuleBuilder objects is defined with a call to the DefineDynamicModule method on the AssemblyBuilder object, a parameter named emitSymbols should be passed as `true`.

2. An item of the type ISymbolDocumentWriter needs to be defined. This is done with a call to a DefineDocument method on the ModuleBuilder object which returns a ISymbolDocumentWriter object. The parameters to this method call include the name of the source file that is being compiled, this is neccessary so that the debugger can prompt for the source file when debugging the compiled file. The ISymbolDocumentWriter object is passed with the CompileContext to all nodes during compilation.

3. Local variables in methods are created with a LocalBuilder object. In a non debug build these locals are not stored by name in the compiled file, but simply given a number and referred to by that number. To be able to map variables in the compiled file to variable names in the source file a method, SetLocalSymInfo, is called on the LocalBuilder object. The method takes

the name of the variable as a parameter and stores that information for later use by the debugger.

4. The method SetUserEntryPoint must be called on the ModuleBuilder object to enable the debugger to know what the entry method of the assembly is. The method takes a MethodBuilder object as a parameter.

5. The most important part of emitting the debug symbols is marking *sequence points* in the CIL bytestream. A sequence point is a point in the bytecode that tells the debugger to stop at that point during code execution and highlight a particular section in the source code file. To be able to do this the sequence point contains information about a start position and end position in the source file, given as line and column numbers. A sequence point is marked with a call to a MarkSequencePoint method on an ILGenerator object, the methods parameters are an instance of ISymbolDocumentWriter and four integers, startLine, startColumn, endLine and endColumn. Figure 3.5 shows a few lines of CIL bytecode interspersed with sequence points. (Note: the CIL file format does not store sequence points in exactly this manner, the figure is simply meant to clarify the concept). It is worth noting that the CIL has no notion of statements, expressions or other programming language constructs, it is perfectly legal to insert a sequence point in the middle of an expression or anywhere else in the bytecode. It is completely up to the programmer to insert sequence points at meaningful points in the bytestream according to the semantics of the language being implemented.

The PLR handles these five steps, so an implementation of a language that uses the PLR as its backend compiler does not need to concern itself with them directly. However, since the PLR does not handle parsing of source files it can not determine itself the line and column numbers needed for marking sequence points. For that purpose, every node in the PLR abstract syntax tree has an instance of a class named LexicalInfo. This class is simply a wrapper around the four integers that a sequence point needs, startLine, startColumn, endLine and endColumn. This information is easily available during parsing and so the individual language parsers should store this information for each node. The PLR will then automatically emit a sequence point before every *action* taken by a process, as well as for expressions evaluated in `if-then-else` processes and for process invocations.

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       22 (0x16)
  .maxstack  8
  IL_0000:  nop
//debugger stops, highlights line 12, col 9-36
[SEQUENCEPOINT (12, 9, 12, 36)]
  IL_0001:  ldstr      "Hello"
  IL_0006:  call       void Program::WriteLine(string)
  IL_000b:  nop
//debugger stops, highlights line 13, col 9-26
[SEQUENCEPOINT (12, 9, 12, 36)]
  IL_000c:  ldc.i4.s   27
  IL_000e:  ldc.i4.4
  IL_000f:  call       void Program::Power(int32, int32)
  IL_0014:  nop
  IL_0015:  ret
} // end of method Program::Main
```

Figure 3.5: CIL bytecode with sequence points

3.4 Runtime Library

The PLR has a runtime library consisting of eleven classes. These classes reside in the PLR.Runtime namespace. Applications compiled using the PLR must have access to this library at runtime in order to execute successfully. Figure 3.6 shows a class diagram of the runtime library. Below is an overview of each of the eleven classes.

ProcessBase is an abstract base class for any compiled processes. It contains methods used to interact with the Scheduler, for instance method to synchronize on channels, methods for startup and termination as well as methods to suspend and resume the process thread. Since processes are built up into a tree-like structure at runtime (further explained in Section 3.5.1) it also contains a field for its parent ProcessBase instance. Finally, it contains a list of IAction instances, this list will hold all actions that occur in the process instance or any of its subprocesses and are restricted by the instances restriction clause.

BuiltIns is a small static class that contains utility methods that can be called by processes, such as a method to print to the console.

IAction is an interface that all runtime actions must implement. It contains four methods. ProcessID returns the id of the process performing the action, IsAsynchronous returns true if the action can be executed without synchronizing with another action, this applies for instance to arbitrary method calls to .NET methods. CanSyncWith(IAction other) determines whether the action can be

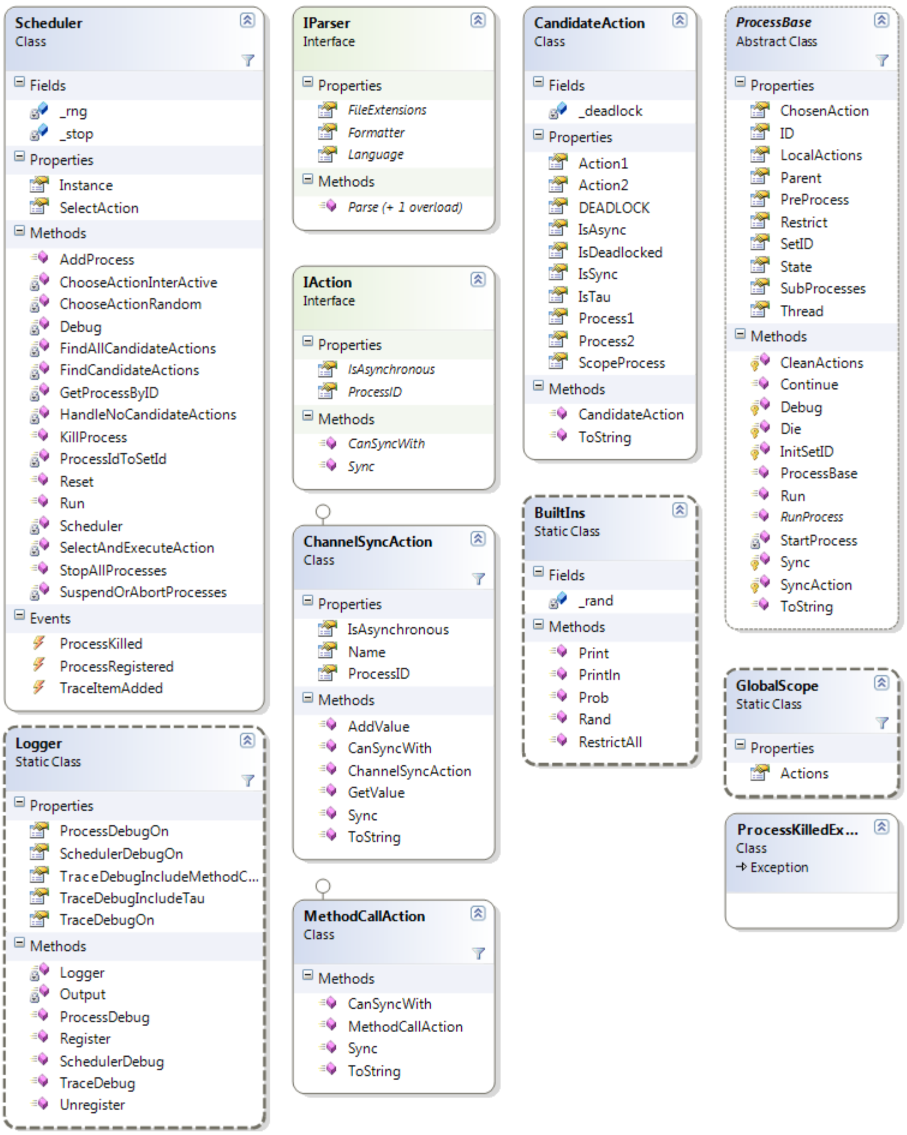


Figure 3.6: PLR runtime library classes

synced with another action, in the case of asynchronous actions this method always returns false. Finally, Sync(IAction other) is called on those actions that have been chosen for execution and is used for example to pass values from one process to another through channels.

IParser is an interface for parsers. It contains Parse methods for streams and files, as well as properties that return the name of the process language the parser is for, the file extensions it uses for source files, and a formatter class that can format a PLR abstract syntax tree and print it out as source code in a particular process language.

ChannelSyncAction is a class representing synchronization on a channel. It contains the channel name, the id of the process performing the action and information about whether the process is attempting to send or receive on the channel. In the case of a send operation it can optionally contain a list of values that are being sent, and in the case of a receive operation it can contain an empty list which values can be copied to during synchronization. Variables are then bound to these values. The ChannelSyncAction implements the IAction interface as all runtime actions must, it is a synchronous action and its CanSyncWith method will only return true for other ChannelSyncAction instance that have the same channel name, are performing the opposing operation and have the same number of values being passed through the channel. When two actions are synchronized, the Sync(IAction other) method on both the actions are called with the other actions as a parameter. In the case of channel synchronizations the action instance which represents the receiving end of the operation will copy the values from the sending action instance to itself. These values are then available to the receiving process which can bind them to its variables. The sending action will do nothing in its own Sync method, as it knows that the receiving action will copy the values over.

MethodCallAction is a runtime action which can be used to call an arbitrary .NET method, either a built-in method from the .NET base class library or a method from any .NET assembly. It is an asynchronous action and as such does not need to synchronize with another process to be executed. Currently the PLR provides support for calling static methods that have integers or strings as parameters. This allows for instance most of the methods from the System.Math class to be accessible. To gain access to instance methods, for example the NextInt method of the System.Random class, it is necessary to write static wrappers around the methods.

Logger is a utility class for handling process output to the screen. Its main feature is assigning a different color to each process to easily distinguish between them in the console output.

GlobalScope is a small class whose only purpose is to be a repository of possible actions that are not restricted by any process. As explained in Section 3.5.1, candidate actions are propagated up the process tree, and at each process it is checked whether the process restricts them, if so they are stored within that process so that they do not synchronize with actions outside the process. In the case where no process restricts the action and it can synchronize with any other action that is not otherwise restricted then the action is stored in the global scope, while it waits to see whether it was chosen for execution.

ProcessKilledException is an exception class used when processes are killed. As an example, when a process is a candidate in non deterministic choice and is not chosen then it must be killed. To bypass all the subsequent actions of the process, a ProcessKilledException is thrown and then caught at the end of the process's code. There the process will unregister itself from the Scheduler, print a message to the console and then terminate.

Scheduler is the real execution engine of the PLR. It follows the *Singleton* [11] design pattern so it is trivial for all processes in the application to gain access to the same Scheduler instance. When processes are activated they register themselves with the scheduler, which keeps a list of active processes. The scheduler then monitors the processes and waits until all processes have generated all their candidate actions and are waiting for an action to be executed so they can continue. At that point the scheduler goes through all the possible actions, figures out which actions can sync with each other and then randomly chooses an action to execute. It then executes the action and wakes up the processes involved in the action so that they can resume execution. It also terminates certain processes (or rather instructs them to terminate themselves). These are generally candidates of non deterministic choice who were not chosen. Once the scheduler has finished one such round it again waits until all the processes it woke up are again suspended and then chooses the next action to execute, and so on. Figure 3.7 shows the workings of the scheduler in pseudocode. Other responsibilities of the scheduler are thread locking and synchronization, and keeping track of the *trace*, that is the list of actions executed during the duration of the program.

CandidateAction is a simple data class used by the scheduler when it is selecting an action to execute. It holds information about a particular action that is ready for execution as well as references to the processes that will perform the action.

3.5 CIL Structure of a Process Language Application

A process language application is in many ways different from an application written in a traditional programming language. One of the goals of this project was to investigate how well process languages are suited to the .NET virtual

```
// active_procs contains all the running processes

do forever:
  all_blocked = true

  for every process p in active_procs
    if state(p) != STATE_BLOCKED
      all_blocked = false

  if all_blocked = true
    //Find matches...
    candidate_matches = []
    for every process p in active_procs:
      //p.restricted_actions contains those actions
      //being performed within p that are restricted
      //by p and so can not go into the global_scope
      //and sync with any other action
      for every action a in p.restricted_actions
        for every action b in p.restricted_actions
          if a can sync with b
            candidate_matches.add( (a,b) )

    for every action x in global_scope
      for every action y in global_scope
        if x can sync with y
          candidate_matches.add( (x,y) )

    if length(candidate_matches) = 0
        DEADLOCK, program finishes
    else
      set match = random(candidate_matches)
      execute(match)
      wake up processes that had actions in the match
      kill processes that were not selected
        in non deterministic choice
```

Figure 3.7: The scheduler algorithm

machine. We now look at how a process language system looks once it has been compiled to the Common Intermediate Language.

3.5.1 Architecture choices

One of the hardest decisions during the design of the PLR was whether or not to represent each process as a separate thread. Writing multi-threaded code is hard, and it is subject to subtle errors, race conditions and other problems that are easily avoided when using only a single thread. However, a single threaded implementation would be forced to represent the processes as datastructures, rather than as independent programs in their own right. That approach, that the processes are datastructures and the program is a single thread operating on those datastructures is certainly worthwhile, and in fact an early prototype was implemented as an interpreter that did just that. It even makes certain things easier, such as the visualizing the state of the processes after each round. However, using multiple threads allowed for compiling each process relatively independently of other processes, and conceptually seemed closer to the semantics of process algebra. Another benefit of the multi-threaded approach was that it made emitting debug symbols fairly simple, while doing the same in a single threaded way would have been problematic. For these reasons the multi-threaded approach was taken.

Another concern when deciding how to structure a process language application was the semantics of restriction and relabeling. When these are applied to a process they keep applying to any subsequent process that it may invoke. E.g. in $(a.P)[d/c]$ the relabeling of c to d has to be applied to everything that happens in the invoked process P. To handle this each process has a reference to the process that spawned it in a Parent property. At runtime P would have $(a.P)[d/c]$ in its Parent property and whenever it performs an action it will first check whether it restricts or relabels it itself, if not it will pass the action up to its parent which can check again if the action is restricted at that level, and and so it propagates up the chain of parent processes. If an action is restricted at a particular level then it can only synchronize with other actions that are at the same level, those actions that are not restricted at all go on up to the global scope, where they become observable from the outside.

The one problem with that approach is that a lot of processes do not restrict or relabel anything at all, and would then be kept alive for no reason, they would simply take up memory and make the process tree unnecessarily complicated. To avoid this, processes only set themselves as the parent of a spawned process if they actually have some restrictions or relabellings. If they do not then they set their own parent as the parent of their spawned processes. This is perhaps best explained by an example.

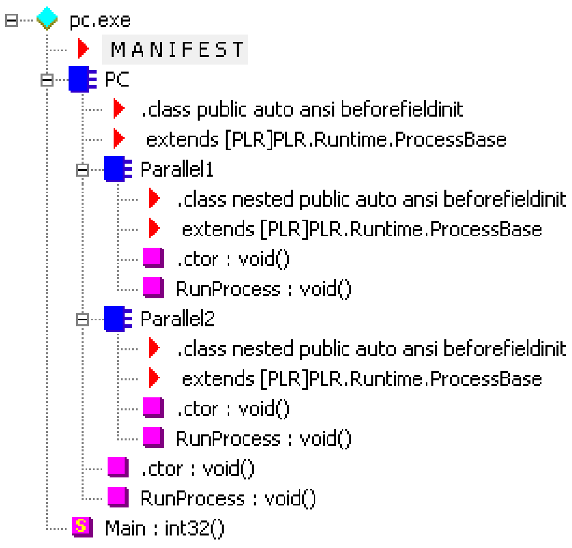
```
A = a . b . B
B = (b . c . C) \{b,c}
C = c . d . D
D = 0
```

In the system above the initial process is A. It performs two actions (another process that synchronizes on these channels is omitted for clarity), and then turns into B. A is not restricted in any way so there is no reason for it to set itself as B's parent. A then checks if itself has a parent, it does not and so it sets B's parent to null before starting it. B on the other hand is restricted, so after it has performed its actions it starts C and sets itself as C's parent. When C performs its c action it is passed up to its parent and is restricted in the B process. When C eventually turns into D it needs to decide what D's parent will be. C itself has no restrictions and so does not need to live on, so it sets D's parent as its own parent, which was B. C can now die and be removed from memory, D has B as its parent so the restrictions of B will continue to be applied correctly.

3.5.2 Processes

Each process in process algebra maps to a class in CIL. The process classes all inherit from a ProcessBase class in the PLR runtime library and override a RunProcess method. When a process has been instantiated, a Run method is called on it, and it will then run the RunProcess method on a new thread.

The top level processes, those that are defined as named process constants, are compiled to classes named after the process constant. A top level class can have multiple inner classes however, and each of those can itself have multiple inner classes. This happens for instance when a process starts by performing an action and then turns into a process that is a parallel composition of other processes, e.g. $a \centerdot (b \centerdot 0 \mid c \centerdot 0)$. In that case each of those parallel processes is an inner class of the original top level process. The same thing happens when a process makes a non-deterministic choice; each of the choices is its own inner class. The reason for making these inner classes was that in a fairly large system the number of classes quickly becomes large, and instead of polluting the top level namespace with dozens of generated class names, they are confined within their owning process. It is also easier to understand what each process represents, the class name PC+Parallel1 represents the first parallel process within the PC process,

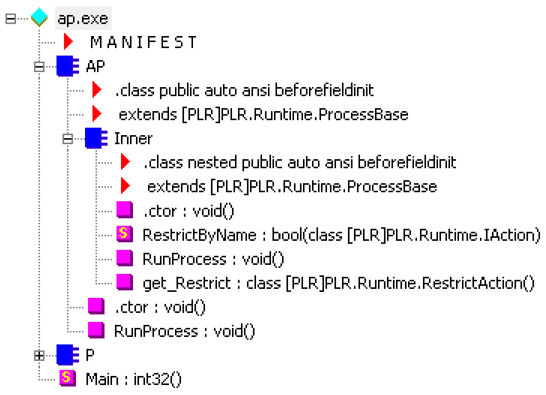


```
pc.exe
    MANIFEST
    PC
        .class public auto ansi beforefieldinit
        extends [PLR]PLR.Runtime.ProcessBase
        Parallel1
            .class nested public auto ansi beforefieldinit
            extends [PLR]PLR.Runtime.ProcessBase
            .ctor : void()
            RunProcess : void()
        Parallel2
            .class nested public auto ansi beforefieldinit
            extends [PLR]PLR.Runtime.ProcessBase
            .ctor : void()
            RunProcess : void()
        .ctor : void()
        RunProcess : void()
    Main : int32()
```

Figure 3.8: Assembly structure of process $PC \overset{\text{def}}{=} a \,.\, (b \,.\, 0 \mid c \,.\, 0)$

which is a parallel composition process. Figure 3.8 shows the assembly structure of a simple parallel composition process, $PC \overset{\text{def}}{=} a \,.\, (b \,.\, 0 \mid c \,.\, 0)$. (This is a screenshot from a tool called ILDASM from Microsoft, the large boxes with three pins represent classes, the triangles represent metadata about the classes and the small rectangles represent methods).

The semantics of action prefixing $(a \,.\, P)$ state that action a is performed and the process then behaves like P. Considering how parallel composition and non deterministic choice were implemented with inner classes it might then seem natural to perform a and then invoke an inner class P. That is not the case however. The reason is that it is very common to have a list of actions performed, e.g. $a \,.\, b \,.\, c \,.\, d \,.\, P$, and creating a new class after every single action would result in a large number of classes for no purpose. So for a process such as this, all the actions are performed in the same class, in its RunProcess method. There are exceptions to this however, if a process performs some actions and then becomes another action prefixed process that is restricted or has relabellings then it will have an inner class at that point. For example the process $a \,.\, b \,.\, (c \,.\, P)\backslash\{c\}$ will perform actions a and b in the main class, but have an inner class for $(c \,.\, P)\backslash\{c\}$. The reason for this is that restrictions and relabellings always have process scope, and since the restriction on c does not apply to the first actions a and b then a new process is needed after a and b



```
ap.exe
    MANIFEST
    AP
        .class public auto ansi beforefieldinit
        extends [PLR]PLR.Runtime.ProcessBase
        Inner
            .class nested public auto ansi beforefieldinit
            extends [PLR]PLR.Runtime.ProcessBase
            .ctor : void()
            RestrictByName : bool(class [PLR]PLR.Runtime.IAction)
            RunProcess : void()
            get_Restrict : class [PLR]PLR.Runtime.RestrictAction()
        .ctor : void()
        RunProcess : void()
    P
    Main : int32()
```

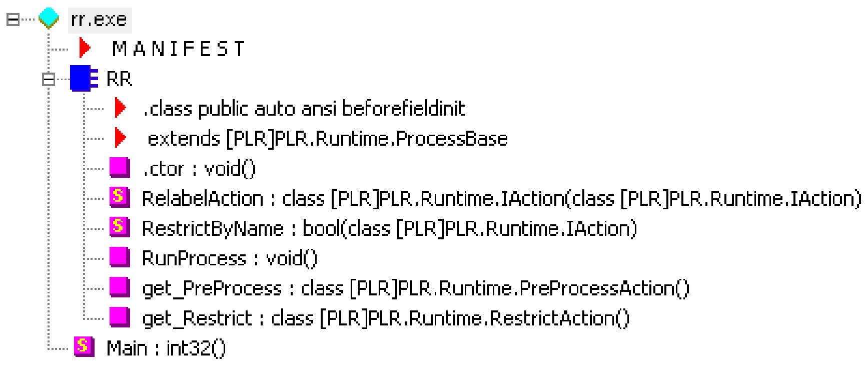
Figure 3.9: Assembly structure of process $AP \stackrel{\text{def}}{=} a \,.\, b \,.\, (c \,.\, P)\backslash\{c\}$

have been performed. Figure 3.9 shows the assembly structure of the process $AP \stackrel{\text{def}}{=} a \,.\, b \,.\, (c \,.\, P)\backslash\{c\}$. Note that the class AP+Inner contains get_Restrict and RestrictByName methods, while the outer class AP does not.

Process invocations, that is when a process turns into a named process, is implemented simply by creating a new instance of the named process and starting it. The process $c \,.\, Proc$ performs action c and then creates a new instance of Proc and starts it, after that has been done $c \,.\, Proc$ simply exits, Proc has been started in its place. The only complication here is if a restriction or relabeling is applied to Proc but not the rest of the process. If the process was written as $c \,.\, (Proc\backslash\{a\})$ then an inner class would be needed, whose only purpose was to create a new instance of Proc and start it. This might seem a wasteful inner class, but keep in mind that this restriction lives on and applies to everything that happens in Proc and any other process that Proc might turn into. The instance of Proc that is started will have a reference to the process $(Proc)\backslash\{a\}$ in its Parent property, any action performed in Proc will be passed up along the parent chain to see if it is restricted or relabeled at some point.

3.5.3 Restrictions and relabellings

Restrictions and relabellings map to methods in CIL. The methods do not have to be member methods of a process class, there is an extra layer of indirection to allow for calling methods in external assemblies. The ProcessBase class has



```
rr.exe
  MANIFEST
  RR
    .class public auto ansi beforefieldinit
    extends [PLR]PLR.Runtime.ProcessBase
    .ctor : void()
    RelabelAction : class [PLR]PLR.Runtime.IAction(class [PLR]PLR.Runtime.IAction)
    RestrictByName : bool(class [PLR]PLR.Runtime.IAction)
    RunProcess : void()
    get_PreProcess : class [PLR]PLR.Runtime.PreProcessAction()
    get_Restrict : class [PLR]PLR.Runtime.RestrictAction()
  Main : int32()
```

Figure 3.10: Assembly structure of process $RR \overset{\text{def}}{=} (a \,.\, b \,.\, c \,.\, 0)_{[x/a,y/b]}\backslash\{c\}$

two methods, get_PreProcess and get_Restrict [1]. These methods return delegates (otherwise known as function pointers) to a preprocess function (such as a relabelling function) and a restriction function, respectively. The base class versions of get_PreProcess and get_Restrict return function pointers to methods that do not alter or restrict any actions. Concrete process classes can override these methods and return function pointers to other methods, either methods in the process class itself, or methods in some external assembly.

The PLR can compile simple restrictions and relabellings directly into the class where they are used. (Simple meaning that they only use constant channel names, like $[a/b]$ or $\backslash\{c, d\}$, and do not require any additional logic). Figure 3.10 shows the assembly structure of a simple restricted and relabeled process, $RR \overset{\text{def}}{=} (a \,.\, b \,.\, c \,.\, 0)_{[x/a,y/b]}\backslash\{c\}$. The restriction method is named RestrictByName and the relabeling method is named RelabelAction. The base class methods get_Restrict and get_PreProcess have then been overriden to return function pointers to RestrictByName and RelabelAction, respectively.

3.5.4 Variables and scope

Variable scope becomes a bit tricky due to the possibly many inner classes of a single process. A simple process like $in(x) \,.\, (\overline{out}(x) \,.\, 0)\backslash\{out\}$ is actually

[1]The odd naming stems from the fact that these are defined as *properties* in the ProcessBase class, where they are named Restrict and PreProcess. Properties in C# simply compile down to getter and setter methods with get_ and set_ prefixed to the property name)

two classes and the *in* and *out* actions are performed in different methods. The variable x still needs to be passed on so that the inner process has access to the value that was bound in the outer process. An additional complication is that the process's main logic has to be defined in an overridden method, RunProcess, so changing the parameters of that method is not an option. And finally, a variable may be bound and the process may then become two or more parallel processes, each of which must have access to the variable. Consider the process $in(x).(\overline{left}(x).0 \mid \overline{right}(x).0)$. Both of the parallel processes use the variable x, but an important thing to note is that they each have their own instance of it. Changing it in one parallel process does not affect it in another. This may seem counterintuitive, but consider if changing the variable in one process did affect it in the other process, then we would have created a new method of communication between processes, shared variables!

The PLR handles variables by passing them as parameters to the inner processes constructor. The inner process then assigns each of the constructor parameters to member variables. The PLR variables are either integers or strings, integers are passed by value and strings are immutable, so there is no danger of two processes changing the same value in memory. At the start of a process's Run-Process method, it defines local variables with the same names as its member variables and assigns the member variables to local variables. This was done for two reasons, it simplifies working with variables in the method since all variable lookups are done on local variables (keep in mind that new local variables might also be defined), and it helps during debugging, the debugger will display the value of local variables when the mouse cursor hovers over their names.

A process may split into many processes and not all of them might need to use all variables. As an optimization, the PLR examines the syntax tree during compilation and only defines member variables and constructor parameters in processes where it is possible that the variable will be used in the process. For example, in the process

$$\text{VAR} \overset{\text{def}}{=} in(x) \,.\, in(y) \,.\, (\overline{out}(x) \,.\, 0 \mid \overline{out}(y) \,.\, 0)$$

the main VAR process will define two local variables, x and y. The first parallel process, $\overline{out}(x) \,.\, 0$, will have a constructor with only one parameter, x and one member variable x, the second parallel process, $\overline{out}(y) \,.\, 0$, will only have y as a member variable and constructor parameter.

3.6 Summary

The Process Language Runtime is implemented as a .NET library, which contains both the abstract syntax tree used during compilation as well as runtime classes used during execution. The syntax tree is the most important part of the PLR, it is a rich datastructure where each node knows how to compile itself and emit the correct bytecodes. Debugging support is also provided by the PLR, although individual language parsers must provide the PLR with information about line and column numbers for the process constructs. Implementers of process algebras can extend the PLR by creating additional syntax tree nodes, subscribing to compilation events and writing their own runtime libraries. When a process language system is compiled then individual processes become classes in .NET, actions become method calls and restrictions and relabellings are implemented as methods, with function pointers used as an abstraction to allow for calling methods in external assemblies.

Chapter 4

Analysis and Optimization

In this chapter we look at the static analyses performed on the PLR abstract syntax tree before compilation, these include some classical dataflow analyses as well as some analyses that are more specific to the process algebra domain. The results of these analyses can be used to optimize the compilation process, the optimizations are presented and explained, and their implications for debugging support are discussed.

4.1 Analysing process algebra

4.1.1 Traditional data flow analysis

Many of the most useful static analyses that compilers perform are based on *data flow analysis*. For these types of analyses it is necessary to build up a *control flow graph* of the program being analysed. The control flow graph shows which program points lead directly to which other program points, in some cases the reverse control flow graph is needed, which shows for a program point p what its immediate predecessors are. The following snippet of typical imperative code is an example:

Example 4.1

```
1: y := 2
2: x := 1
3: while x < 6 do
4:    x := x + 1
   od
5: print x
```

In this example the program points are labelled from 1-5. Program points are those points in the program where something happens, expression are evaluated, variables are assigned, etc. The control flow for this example would be (1,2), (2,3), (3,4), (3,5) and (4,3). The analysis is then typically performed by having each program point have an input set and an output set, the input set represents the state of the program as the point is reached and is based on the state of its predecessors, the output set represents the state of the program after the program point has been evaluated, and is based on the points input set with some modifications based on what happened at the program point.

To give a concrete example suppose we have an analysis which is determining for the code snippet in Example 4.1 which variables have been assigned at each point in the program. For the program point labeled 2 (x := 1), its input set would be $\{y\}$ as y is the only variable that has been assigned at that point. Since program point 2 assigns to the variable x then its output set would be the union of its input set and $\{x\}$, or $\{x, y\}$. The functions used to modify the input set to create the output set are commonly called *Kill* and *Gen*, the *Kill* function removes items from the input set and the *Gen* function adds new items to the output set. This can be shown as (for program point p):

$$p_{output} = p_{input} \ \cup \ Gen(p) \ \setminus \ Kill(p)$$

To get the final result of the analysis these calculations must be repeated for each point in the program until the input and output sets of each become stable. The result is an approximation, either over approximation (something *may* happen on the path to the program point) or an under approximation (something *must* have happened on the path to the program point). To calculate the input and output sets of each program point an iterative worklist algorithm is used. There are many variations of these algorithms, they keep track of which program points change and which other program points must then be re-calculated. There is a lot more to data flow analysis than explained here, for instance whether output sets of predecessors are combined using the union or intersection operator, and what the initial content of the input sets are, but we will not go into more detail on how data flow analysis generally works here, this is meant only as a brief introduction before explaining the analyses performed in the PLR. A more formal introduction to the subject of data flow analysis can be found in [27].

4.1.2 Data flow analysis in process algebra

Two properties of process algebra make it different from imperative languages when it comes to data flow analysis:

1. There are no looping constructs. The reason that data flow analysis on imperative languages needs iterative worklist algorithms is because of looping. When looping is not a part of the language then the whole program can be analyzed from top to bottom (or bottom to top) in one pass, each program point only needs to be calculated once.

2. There are branches, but they are never re-joined later in the program. This implies that at every point in the program is is possible to know exactly what path was taken to get to that point. Note that this only holds for forward analysis. Backwards analysis can be seen as multiple branches joining, and so in a backward analysis it is not possible to know at program point p from which branch a particular item in p's input set originates.

These two properties might not hold for all process algebras in existence, but they certainly hold for both CCS and KLAIM, as well as some other prominent algebras such as CSP, and they do simplify the implementation of these analyses for process algebra. Sections 4.2.1 and 4.2.2 discuss one backward and one forward data flow analysis and how they were implemented in an extensible way for the PLR.

4.2 Analyses

The analyses presented here generally follow the same pattern. They implement an IAnalysis interface and inherit from AbstractVisitor. They traverse the syntax tree to analyze it and issue warnings about any anomalies found. In some cases they mark certain syntax nodes as not being used, the compilation then uses that information for optimization, for example to skip compiling certain things that are guaranteed never to be executed. These optimizations can be turned on and off in the compile options that are passed to the PLR. The optimizations and debugging support are mutually exclusive, this is due to the fact that the sequence points in an optimized executable may not match the actual source code file any longer, which could make the debugging rather confusing.

4.2.1 Live Variables

Live Variables Analysis is a classic data flow analysis. Its purpose is to identify at each program point which variables are *live*, that is which variables will be read later on in the program in the paths that follow the program point in question. This is a backward analysis and is traditionally used to identify assignments to variables that have no effect, for instance if the variable x is assigned at program point p, but along all paths that follow p the variable is either never read, or assigned to again before being read then the assignment at p has no effect and can be eliminated. The analysis is an over-approximation, we cannot safely eliminate the assignment to x if there *may* be a path after p where x is read. This implies that the input set of p will be the *union* of the output sets of its pre-decessors (here the pre-decessors actually mean the program points that follow p as it is a backward analysis).

In the analysis of the PLR syntax tree, a process is considered a program point. Additionally, syntax nodes that represent a process constant being defined are considered program points, as they may contain defining occurrences of variables. For instance, in $CS(x, y) \stackrel{\text{def}}{=} a.0$ we would consider $CS(x, y)$ a program point, since it defines the variables x and y, and it can be beneficial at that point to know whether the initial value of those variables was ever used in the process. We have two helper functions, *assigned(x)* takes in an action and returns a set of the variables assigned in that action. An action in this case can be the receiving of values through a channel or sending values through a channel. The other helper function, *read(x)* takes in an expression and returns all variables evaluated in that expression, or takes in an action and returns all variables evaluated in that action. Figure 4.1 shows some examples of the use of *read* and *assigned*, Figure 4.2 shows how the *Kill* and *Gen* functions are defined for each of the process types in the PLR syntax tree.

$$\begin{aligned}
\textit{assigned}(\text{channel}(x, y)) &= \{x, y\} \\
\textit{read}(\overline{(\text{channel}(z + 1, a - b)})) &= \{z, a, b\} \\
\textit{read}(x + 1/z) &= \{x, z\}
\end{aligned}$$

Figure 4.1: Examples of the *read* and *assigned* functions

$$
\begin{array}{rcl}
Kill(a.P) & = & assigned(a) \\
Gen(a.P) & = & read(a) \\[6pt]
Kill(P \mid Q) & = & \emptyset \\
Gen(P \mid Q) & = & \emptyset \\[6pt]
Kill(P + Q) & = & \emptyset \\
Gen(P + Q) & = & \emptyset \\[6pt]
Kill(0) & = & \emptyset \\
Gen(0) & = & \emptyset \\[6pt]
Kill(\text{if } bexp \text{ then } P \text{ else } Q) & = & \emptyset \\
Gen(\text{if } bexp \text{ then } P \text{ else } Q) & = & read(bexp) \\[6pt]
\text{(process definitions)} & & \\
Kill(K(x, y, z) \overset{\text{def}}{=}) & = & \{x, y, z\} \\
Gen(K(x, y, z) \overset{\text{def}}{=}) & = & \emptyset \\[6pt]
\text{(process invocations)} & & \\
Kill(K(exp_1, ..., exp_n)) & = & \emptyset \\
Gen(K(exp_1, ..., exp_n)) & = & read(exp_1) \cup ... \cup \ read(exp_n)
\end{array}
$$

Figure 4.2: *Kill* and *Gen* functions for Live Variables

The actual implementation of the *assigned()* and *read()* functions is done with two properties that are present on all syntax nodes that derive from either Process or Action. These properties are AssignedVariables and ReadVariables and return the assigned and read variables of a process or action. Having these as properties of the syntax nodes instead of as part of the implementation of the analysis allows for greater extensibility. Languages implemented using the PLR that have their own custom syntax nodes simply need to override these properties and can then use the analysis without further modifications. This is the case in the KLAIM implementation discussed in Chapter 6, it has its own custom actions and a custom process type and they simply implement these properties.

Constructing the flow graph is trivial, since there are no loops or joining branches. In fact, a special flow graph is not constructed, instead the PLR syntax tree is used directly. To make it simpler, each Process node is required to implement a FlowsTo property which is a list of all the processes it can flow to. For an `if-else` process these are the if and else branches, for parallel composition these are the composed processes, etc. In addition each node in the syntax tree has a Parent property which references its parent and can be used as a reverse flow graph. Again, having these properties directly on the syntax tree is useful so that additional process types can be added without having to modify the analysis code. The input and output sets are stored in a a Tag property which is also on each syntax node, this is a generic object reference which analyses may use to temporarily store data associated with each node.

Instead of an iterative worklist algorithm the analysis itself uses the visitor pattern discussed in Section 3.3.2. It simply inherits from AbstractVisitor and overrides the Visit(Process p) method which ensures that it will be called for every node in the syntax tree that inherits from Process. The visitor does a depth-first recursive traversal of the syntax tree, the Visit is called on child nodes before parent nodes so whenever we are processing a process p we know that all its child nodes have already been processed. What is needed in the method itself is then:

1. Construct the input set of p by joining the output sets of all the processes in its FlowsTo property.
2. Check whether any variable x in p.AssignedVariables is not in p's input set. If it is not, then the assignment at p has no effect and a warning is issued, and an IsUsed property on x is set to `false`. This can be used later for optimization during compilation.
3. Construct the output set of p by taking the input set and removing all variables that are in p.AssignedVariables and then adding all variables that are in p.ReadVariables.

Once the analysis has been performed a number of warnings have been issued and all assignments to variables that have no effect have been marked as not used. In some cases it may be needed to assign to a variable that is never used because the interface of another process requires it. For instance, one process may send a value on channel ch, another process may wish to synchronize but does not care about the value sent. In that case the variable that is bound during the synchronization can be named **notused** and then no warnings will be issued.

During the compilation stage the PLR will look at every variable that is assigned and check whether its IsUsed property is set to `false`. If the assignment is never used then no bytecode is emitted to store a value in the variable and no variable is declared. If the assignment is unused because another assignment is made before the variable is read, then the variable is defined at that point instead of the original point. Due to the way the PLR is constructed this may save one or more processes from having that variable as a field or a local variable, thus reducing the size of the compiled executable.

4.2.2 Reaching Definitions

Reaching Definitions is another classic data flow analysis. Its purpose is to identify which assignments may reach a given point in a program.

Example 4.2

```
1: x := 1
2: x := 2
3: print x
```

In the example above the assignment at line 2 reaches line 3. The assignment at line 1 reaches line 2 but not 3, since x is assigned again at line 2. So at line 3 we know that the value of x is that which was assigned at line 2. With branching and re-joining branches there may be more than one definition of a particular variable that reach a certain point, for example when a variable is assigned in both branches of an `if-then-else` statement. However since this is a forward analysis and the PLR does not have re-joining branches this cannot happen in the PLR's implementation of this analysis.

Reaching Definitions analysis can be used for many common compiler optimizations, such as constant propagation and common subexpression elimination. However in the PLR it is used for only one thing, to detect the use of unassigned variables. Once the analysis has been performed it is possible to examine each program point p and see whether all variables read at p have entries in p's input set. If a variable x is evaluated at p and does not exist in p's input set, then there is no definition of x that reaches p and it is being used before being assigned. The *Kill* and *Gen* functions are shown in Figure 4.3. Since the assignment to variable x defined at one point in the program is not the same as the assignment to variable x defined at another place in the program the function *named()* is used. $Kill(a.P) = (named(assigned(a)))$ means to remove all the variables from the set which are named the same as those that were assigned in a, whereas for the *Gen* we have $Gen(a.P) = assigned(a)$ which means to add the exact variables occuring in a to the set.

The implementation of Reaching Definitions is similar to that of Live Variables. An analysis class inherits from AbstractVisitor and overrides its Visit(Process p) method. Since this is a forward analysis the traversal is a little different than in Live Variables. Here, the property VisitParentBeforeChildren is set to `true` on the AbstractVisitor. This makes sure that the tree can be processed in forward order instead of backward. Then, in the method the following steps are performed:

1. Check the input set of p against the property p.ReadVariables. If there

$$\begin{aligned}
Kill(a.P) &= named(assigned(a)) \\
Gen(a.P) &= assigned(a) \\[1ex]
Kill(P \mid Q) &= \emptyset \\
Gen(P \mid Q) &= \emptyset \\[1ex]
Kill(P + Q) &= \emptyset \\
Gen(P + Q) &= \emptyset \\[1ex]
Kill(0) &= \emptyset \\
Gen(0) &= \emptyset \\[1ex]
Kill(\text{if } bexp \text{ then } P \text{ else } Q) &= \emptyset \\
Gen(\text{if } bexp \text{ then } P \text{ else } Q) &= \emptyset \\[1ex]
\text{(process definitions)} & \\
Kill(K(x, y, z) \overset{\text{def}}{=}\ ) &= named(x, y, z) \\
Gen(K(x, y, z) \overset{\text{def}}{=}\ ) &= \{x, y, z\} \\[1ex]
\text{(process invocations)} & \\
Kill(K(exp_1, ..., exp_n)) &= \emptyset \\
Gen(K(exp_1, ..., exp_n)) &= \emptyset
\end{aligned}$$

Figure 4.3: *Kill* and *Gen* functions for Reaching Definitions

exists a variable in p.ReadVariables that has no entry in p's input set then it is being used before assignment and a warning is issued.

2. Create the input set of each process P_i in p.FlowsTo by taking the input set of p and adding all variables from p.AssignedVariables, replacing any previous entries for the same variable.

After the analysis has run the PLR checks whether it produced any warnings. If it did then an error message to that effect is printed to the screen and the compilation is aborted.

4.2.3 Constant Expressions

A simple optimization that can be performed is to calculate the expressions that can be fully evaluated at compile time and replace them with a constant with the result of the calculation. Granted, it is unlikely that large expressions can be computed at compile time, unless some constant propagation is also performed, but it is nevertheless useful to compute those simple ones that can. The implementation is simple (and was partially shown in Figure 3.3), simply subclass an AbstractVisitor and override a Visit method for each of the binary expression nodes (arithmetic, relational and logical). Check if both the left and right childnodes are constants, if they are then calculate the result according to the operator stored in the binary node. Finally replace the binary node with a new constant node that carries the result of the computation. The same goes for the unary minus node.

The one extra thing that this analysis does is that it also visits the BranchProcess node, which is the implementation of the `if-then-else` statement. If the boolean expression in the BranchProcess has been fully computed then the analysis will set the IsUsed flag to false on the branch that will never been chosen, according to the result of the boolean expression. During the compilation stage, if optimizations are turned on, the PLR will then replace the BranchProcess with the branch that is guaranteed to be taken and only compile that branch.

4.2.4 Nil Process Warnings

Nil Process Warnings is a simple analysis which looks for nil processes in places where they are useless and can be eliminated. This includes nil processes that are part of parallel composition ($P \mid 0 \equiv P$) and nil processes that are an option in non deterministic choice (in $P + 0$ the nil process will never be selected). This is implemented as a visitor that visits all NilProcess nodes, checks whether their parent nodes are ParallelComposition or NonDeterministicChoice. If they are, then a warning is issued and the property IsUsed on the nil process nodes is set to `false`, which allows the compilation to skip those processes.

The analysis also checks for process definitions that are defined as nothing more than the nil process (e.g. $P \overset{\text{def}}{=} 0$) and issues a warning that states that any process invocation of P can be replaced with a nil process.

4.2.5 Unmatched Channels

When a process tries to synchronize on a channel that no other process anywhere in the application ever synchronizes on then that process will be blocked forever. That scenario is not hard to detect manually in a small system, however once the system grows it becomes increasingly harder, especially when we factor in that a channel can be relabeled multiple times.

To attempt to detect these synchronizations which will never complete, the PLR contains an analysis named *Unmatched Channels*. It begins by collecting the channel names of all InAction and OutAction instances in the system. It then collects all the relabellings that can occur (although it does not take into account relabeling that is done by custom .NET methods). Every variation of channel names that are possible are then created using the relabellings, this is done on a global scale without consideration for whether the relabeling has a chance of actually being applied to a particular channel. Once all the variations have been created the input channels are compared to the output channels, and if there is an input channel that has no corresponding output channel, or vice versa, then a warning is issued that the process will block forever should it attempt to synchronize on that channel.

Actions can only be child nodes of ActionPrefix nodes, if an action is guaranteed to block forever, its parent ActionPrefix node is gotten and the process that follows the action has its IsUsed property set to `false`. This allows the PLR to skip compiling the following process entirely, as it is guaranteed never to run.

Clearly this analysis is an imprecise approximation of the problem itself. There could be channels that block that the analysis does not detect, because it has found a relabeling that causes it to believe that the channel can synchronize, even if the relabeling actually has no chance of being applied to that particular channel. Even so, the analysis does detect problems in process algebra systems, and is useful for example when channel names or relabellings have been mistyped.

It would be interesting to further refine this analysis, for example by traversing through the tree and figuring out exactly which relabeling could potentially be applied to which channels. Another possible refinement would be to detect cases such as $P = (ch.0 + \overline{ch}.0)$. Here the input and output channel both exist, however since they are part of the same nondeterministic choice only one of them can be chosen and so they will never synchronize with each other.

4.2.6 Unused Processes

Unused Processes is an analysis that traverses the syntax tree and collects the names of all defined processes, as well as all instances where processes are invoked. It then proceeds to check whether all defined processes are invoked at least once. If a process is never invoked a warning is issued to that effect, and if optimizations are turned on the unused process is never compiled at the compilation stage, giving some space savings in the compiled executable. This is a simple optimization but can give significant space savings if large processes are unused. Of course if the compiled assembly is supposed to be used by another assembly then this optimization should not be used, as the analysis cannot detect whether outside callers intend to use a specific process.

4.3 Summary

The Process Language Runtime contains a number of analyses and optimizations that are performed before and during compilation. These analyses are somewhat different from analyses of imperative programming languages since process algebra has a simpler control flow graph which is always a tree. The analyses find assignments to variables that are never used, variables that are read before being assigned to, constant expressions, nil processes that have no effect, synchronizations that can never happen and processes that are never used. The optimizations that can be performed as a result of these analyses are mainly focused on removing code that either has no effect or can never be executed.

Chapter 5

CCS Implementation

In this chapter we look at the Calculus of Communicating System process language, its history, formal syntax and semantics, and its implementation with the PLR as the back-end compiler.

5.1 Overview

CCS was created by Robin Milner in the 1970's where it was the subject of a number of papers [19, 20, 21] and finally a book of the same name [22] published in 1980. It is, along with CSP, one of the oldest process algebras, but is still widely taught in university courses about concurrent systems and is the language of choice in textbooks on concurrent systems such as [1].

In its pure form the language is fairly simple and contains fewer constructs than some of the other process algebras which came after it. This makes it an ideal candidate to use as the blueprint for the PLR. The PLR and the CCS compiler were developed in parallel and the PLR provides all the constructs that CCS needs. Because of that the CCS compiler itself is fairly small and mostly has to do with the front-end, that is the lexer and parser. A description of another language implementation, one that extends the PLR further, is the subject of Chapter 6.

5.2 Syntax and semantics

The version of CCS defined here is equivalent to the one defined in [1], other variants sometimes have slightly different syntax (e.g. use nil instead of 0). The language can be described by the following syntax

- Let $\mathcal{A}$ be the set of *channel names*, we use α as a typical member of this set in demonstrations.
- $\overline{\mathcal{A}} := \{\overline{a} \mid a \in \mathcal{A}\}$ denotes the set of co-names. For each channel name there is a corresponding co-name. Channel name represents input, co-name represents output.
- Let $\mathcal{L} := \mathcal{A} \cup \overline{\mathcal{A}}$ be the set of *labels*.
- $\mathsf{Act} := \mathcal{L} \cup \tau$ is the set of *actions*, where τ denotes the silent (or unobservable) action.
- Let $\mathcal{K}$ be a set of process identifiers.
- The set $\mathcal{P}$ of process expressions is defined by the following syntax:

$$\begin{array}{lcll} P & ::= & 0 & \text{(Nil process)} \\ & \mid & \alpha\ .\ P & \text{(Action prefixing)} \\ & \mid & K & \text{(Invoking a process)} \\ & \mid & P \mid Q & \text{(Parallel composition)} \\ & \mid & P + Q & \text{(Nondeterministic choice)} \\ & \mid & P[f] & \text{(Relabelling)} \\ & \mid & P \setminus L & \text{(Restriction)} \end{array}$$

  where

  - P, Q are processes in $\mathcal{P}$
  - α is an action in Act
  - K is a process name from $\mathcal{K}$
  - $f : \mathsf{Act} \to \mathsf{Act}$ is a relabelling function satisfying the constraints $f(\tau) = \tau$ and $f(\overline{a}) = \overline{f(a)}$ for each label a.
  - L is a set of labels from $\mathcal{L}$

- We assume that the behavior of each process name $K \in \mathcal{K}$ is given by a defining equation

$$K \overset{\text{def}}{=} P$$

where $P \in \mathcal{P}$.

To avoid the use of too many parantheses in writing CCS expressions the convention is used that operators have decreasing binding strength in the following order: restriction and relabelling (the tightest binding), action prefixing, parallel composition and summation. For example, the expression $a.0 \mid b.P \setminus L + c.0$ stands for

$$((a.0) \mid (b.(P \setminus L))) + (c.0)$$

For an informal description of each of the constructs (action-prefixing, parallel composition, non-deterministic choice, restriction and relabelling) we refer to Section 2.1.2 where such a description was given. For a formal description, Figure 5.1 shows the structural operational semantics of CCS.

5.3 Value passing CCS

The syntax and semantics given in Section 5.2 above are those for so-called *pure* CCS, in which communication is pure synchronization and involves no exchange of data. A more practical approach would be to allow processes to send and receive data when communicating on channels, this is a great convenience when modelling certain types of processes. R. Milner, the originator of CCS introduced an extension to CCS in [23], *value-passing CCS*. In the same book he proved that while the extension was convenient, it was theoretically unnecessary. Value-passing CCS introduced three main things which contribute to efficently modelling systems that handle data.

1. Data can be sent and received through channels during synchronization. On the receiving end the data is bound to a variable name. Below we see the value 5 being passed from process Q to process P through the channel a.

   Example 5.1

$$\mathrm{P} \overset{\mathrm{def}}{=} a(x) \,.\, 0$$
$$\mathrm{Q} \overset{\mathrm{def}}{=} \overline{a(5)} \,.\, 0$$

$$\text{ACT} \quad \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

$$\text{SUM}_j \quad \frac{P_j \xrightarrow{\alpha} P'_j}{\sum_{i \in I} P_i \xrightarrow{\alpha} P'_j} \quad \text{where } j \in I$$

$$\text{COM1} \quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

$$\text{COM2} \quad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\text{COM3} \quad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\text{RES} \quad \frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L} \quad \text{where } \alpha, \overline{\alpha} \notin L$$

$$\text{REL} \quad \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

$$\text{CON} \quad \frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'} \quad \text{where } K \overset{\text{def}}{=} P$$

Figure 5.1: CCS structural operational semantics

The output action can contain variable names, constant values or arithmetic expressions, e.g. $\overline{a(x + y/4)}$. The input action can only contain variable names to bind to the incoming value(s).

2. Process constants can be parameterized. When a process turns into a parameterized process, it passes a value to that process, which in turn binds the value to the variables specified in the process definition. Example 5.2 shows a process P that syncs on channel a, then turns into process Q and passes it the value 4. When process Q starts it has the value 4 bound to the name x and tries to pass that to some other process through the b channel.

$$\text{INP} \quad \frac{}{a(x).P \overset{a(n)}{\rightarrow} P[n/x]}$$

$$\text{OUTP} \quad \frac{}{\overline{a}(e).P \overset{\overline{a(n)}}{\rightarrow} P} \quad n \text{ is the result of evaluating } e.$$

$$\text{COND1} \quad \frac{P \overset{\alpha}{\rightarrow} P'}{\textbf{if } bexp \textbf{ then } P \textbf{ else } Q \overset{\alpha}{\rightarrow} P'} \quad \text{bexp is true}$$

$$\text{COND2} \quad \frac{Q \overset{\alpha}{\rightarrow} Q'}{\textbf{if } bexp \textbf{ then } P \textbf{ else } Q \overset{\alpha}{\rightarrow} Q'} \quad \text{bexp is false}$$

$$\text{PCONST} \quad \frac{P[v_1/x_1, ..., v_n/x_n] \overset{\alpha}{\rightarrow} P'}{\mathrm{A}(e_1, ..., e_n) \overset{\alpha}{\rightarrow} P'} \; \mathrm{A}(x_1, ..., x_n) \overset{\text{def}}{=} P \text{ and each } e_i \text{ has value } v_i$$

Figure 5.2: Value-passing CCS structural operational semantics

Example 5.2

$$\begin{aligned} \mathrm{P} &\overset{\text{def}}{=} a \,.\, \mathrm{Q}(4) \\ \mathrm{Q(x)} &\overset{\text{def}}{=} \overline{b(x)} \,.\, 0 \end{aligned}$$

3. A conditional expression is introduced into the language. It takes the form **if** bexp **then** P **else** Q. Example 5.3 shows a process P which checks whether the value of variable x is less than 2, if so it turns into process Q, otherwise it turns into the nil process and terminates.

Example 5.3

$$\mathrm{P(x)} \overset{\text{def}}{=} \textbf{if } x < 2 \textbf{ then } Q \textbf{ else } 0$$

The structural operational semantics for these additions to the language are given in Figure 5.2.

5.4 Implementation

The CCS implementation was written in C# using Visual Studio 2008 as the development environment. It implements both pure and value-passing CCS, it is allowed but not necessary to pass values when synchronizing on channels. Since the PLR includes abstract syntax tree nodes for all constructs of CCS, no additional nodes were created specifically for the CCS implementation. There are only a handful of classes used, below is a short summary of each one.

5.4.1 Class overview

- Program is the entry point into the compiler. It contains a Main method that parses command line parameters, validates parameters such as input and output file names and then creates a Parser object. It then calls the parser's Parse method, receives a PLR ProcessSystem node and calls Compile on it.

- Scanner is the lexer class whose responsibility it is to tokenize a CCS source code file into valid CCS terminals. Figure 5.3 shows the more complicated terminals of CCS defined by regular expressions. The simpler terminals, who are just string constants, are given directly in quotes in the parser definition. The scanner is implemented as a finite state automaton.

- Parser is a recursive-descent parser for CCS. It uses the tokens generated by Scanner and applies a number of productions to recognize the language. The Extended Backus-Naur Form (EBNF) of the productions is given in Figure 5.4. The parser constructs a PLR abstract syntax tree as it parses, if the parsing is without errors the syntax tree can then be compiled. Both the Parser and Scanner are generated by the parser generator Coco/R [25]. It takes as input an EBNF specification of the language interspersed with C# source code and from that it generates the Scanner and Parser classes.

- ParserService is a simple class that implements the PLR's IParser interface to make the parser discoverable from outside the assembly. It implements the interface's methods and delegates the actual parsing to the Parser class.

- ParserTest contains unit tests for the parser and scanner, to be run with the NUnit unit testing framework.

```
PROCNAME    = [A-Z][A-Za-z0-9]*
LCASEIDENT  = [a-z][A-Za-z0-9]*
CLASSNAME   = [A-Z][A-Za-z0-9]*(\\.[A-Z][A-Za-z0-9]*)*
OUTACTION   = _[a-z][A-Za-z0-9]*_
METHOD      = :[a-zA-Z][A-Za-z0-9]*
NUMBER      = [0-9]+
STRING      = "[^"]*"'
```

Figure 5.3: Terminals of CCS scanner

5.4.2 Extensions to CCS syntax

Section 5.2 shows the formal syntax for CCS, it however does not account for integrating with the .NET environment to allow arbitrary method calls to be made as actions and .NET methods to be used as relabelling functions and/or restriction functions. The EBNF specification for the parser which shows all allowed syntax can be seen in Figure 5.4, but to quickly summarize the changes from formal CCS, they are as follows:

- A CCS source code file can start with one or more `use` statements, which consist of the token `use` followed by the fully qualified name of a .NET class. E.g.

  ```
  use System.Console
  ```

  These referenced classes can be in the .NET core library, `mscorlib`, or in any arbitrary .NET assembly. During compilation the filenames of assemblies containing classes used in the source code must be passed to the compiler so that it knows where to look for classnames found in `use` statements.

- Actions can be calls to .NET methods in addition to synchronization and value passing on channels. A .NET method call consists of a colon followed by the method name and parantheses around expressions passed as parameters to the method, e.g.

  ```
  P = :Print("Hello world") . 0
  ```

  At compile time the PLR resolves which class the method belongs to by looking at the classes imported with `use` statements and inspecting their methods. If more than one imported class has a candidate method an exception is thrown. The methods must be static, if instance methods are to be used it is necessary to write a static wrapper method around them that creates an instance of the object in question and calls the instance method.

```
CCS = { ClassImport } ProcessDefinition { ProcessDefinition }

ClassImport = "use" CLASSNAME

ProcessDefinition =
  PROCNAME [ "(" LCASEIDENT {"," LCASEIDENT } ")" ] "=" Process

Process = NonDeterministicChoice

NonDeterministicChoice = ParallelComposition { "+" ParallelComposition }

ParallelComposition = ActionPrefix { "|" ActionPrefix } .

ActionPrefix =
  { Action "." }
  (
    "(" Process ")"
    | "0"
    | ProcessConstantInvoke
    | BranchProcess
  )
  [ Relabelling ]
  [ Restriction ]

BranchProcess = "if" Expression "then" Process "else" Process .

ProcessConstantInvoke =
  PROCNAME [ "(" ArithmeticExpression {"," ArithmeticExpression } ")" ]

Action =
  LCASEIDENT [ "(" LCASEIDENT { "," LCASEIDENT } ")" ]
  | OUTACTION [ "(" ArithmeticExpression { "," ArithmeticExpression } ")" ]
  | METHOD "(" [ CallParam { "," CallParam } ")"

CallParam = ArithmeticExpression | STRING

Relabelling =
  "[" METHOD "]"
  | "[" LCASEIDENT "/" LCASEIDENT { "," LCASEIDENT "/" LCASEIDENT } "]"

Restriction =
  "\"
  (
    LCASEIDENT
    | "{" LCASEIDENT {"," LCASEIDENT } "}"
    | METHOD
  )

Expression = OrTerm { "or" OrTerm }

OrTerm = AndTerm { "and" AndTerm }

AndTerm = RelationalTerm { "xor" RelationalTerm }

RelationalTerm =
  ArithmeticExpression [ ("=="|"!="|">"|">="|"<"|"<=") ArithmeticExpression ]

ArithmeticExpression = PlusMinusTerm { ("+" | "-") PlusMinusTerm }

PlusMinusTerm = UnaryMinusTerm { ("*"|"/"|"\%") UnaryMinusTerm }

UnaryMinusTerm =
  [ "-" ]
  (
    "(" ArithmeticExpression ")"
    | NUMBER
    | "0"
    | "true"
    | "false"
    | LCASEIDENT
  )
```

Figure 5.4: EBNF Productions of CCS parser

- Relabelling functions can be specified as .NET methods in addition to being constant replacements. To use a .NET method for relabelling, the methods name prefixed with a colon is put inside the square brackets that usually define relabelling functions in CCS, e.g. `[:MyRelabelMethod]`. The method is resolved to an imported class at compile time and is required to be a method that takes a single parameter, an instance of the IAction interface from the PLR runtime library.

- Restrictions functions can be specified as .NET methods in much the same way as relabelling functions and must resolve to a method that takes an IAction instance as a parameter. An example of a process which uses a .NET method for restriction could be `(a . 0) \ :MyRestrict`.

5.5 CCS example system

To get a better understanding of what CCS can be used for and how all the parts discussed previously fit together we now look at a larger example. Figures 5.5 and 5.6 show a model of an automatic teller machine, an ATM. The example is written in CCS source code which is slightly different from the formal CCS syntax used in the previous examples. The main differences is that output on channels is represented as a channel name surrounded by underscores, e.g. *_channel_* instead of with an overline, $\overline{channel}$. Other changes from the formal syntax are trivial and will become obvious when looking at the example.

The example uses most of the features of the CCS implementation, we now look further at each of the processes in the example system and discuss the noteworthy aspects of their implementation.

- Example is the root process. The first process in a CCS file is always considered the root process and the only one that is instantiated at the beginning of a program. Its responsibility is to start up other processes in the system, and it does that by becoming a parallel composition of John which is a customer and ATM which represents the ATM machine.

- The ATM is the ATM machine itself. We see that it does not do much by itself, instead it is composed of three component processes running in parallel. They are the UI process which is the interface to the user, the Bank and a CashDispenser component. The ATM uses *restriction* to hide a number of internal events of these component processes, so that the customer cannot directly synchronize with them.

- The Bank process is made up of components as well, these are the Accounts process and the PinCheck process which run in parallel.

- Customer is a parameterized process with parameters for card number, pin number and amount to withdraw from the ATM. John is simply an instantiation of the Customer process with specific values for these parameters. The Customer process uses non deterministic choice to handle error conditions, such as the pin number being incorrect, the ATM being empty or the bank account not having sufficient funds for the withdrawal. Since the ATM will only offer one synchronization at any point the customer will have to take the path dictated by the ATM, e.g. after entering the pin the ATM might output on *wrongpin*, which forces the customer to accept that since it cannot perform the *withdraw* synchronization if the ATM does not allow it. Generally the successful path is the first path in each choice, and the following paths represent error messages. Another thing to notice is that the Customer process uses the built in function

```
use PLR.Runtime.BuiltIns
use Bank.Functions

Example = John | ATM

ATM = (UI | Bank | CashDispenser) \{checkpin, validpin, invalidpin,
                                    accwithdraw, accwithdrawn,
                                    accnotenoughbalance,
                                    checkavailablecash, enoughcash,
                                    notenoughcash,dispensecash}

Bank = PinCheck | Accounts

John = Customer(1234, 3321, 20000)

Customer(cardnr, pinnr, amount) =
   _card_(cardnr) . _pin_(pinnr) .
   (
      _withdraw_(amount) .
      (
         cash(received) . card . :Print("Successful withdrawal") .0
         +
         notenoughmoney . card . :Print("Not enough balance") . 0
         +
         atmempty . card . :Print("ATM is empty") . 0
      )
      +
      wrongpin . card . :Print("Wrong pin number") . 0
   )
```

Figure 5.5: ATM system example in CCS, part 1

```
UI = card(cardnr) . pin(pinnr) . _checkpin_(cardnr,pinnr) .
     (
        validpin . withdraw(amount) . _checkavailablecash_(amount) .
        (
           enoughcash . _accwithdraw_(cardnr,amount) .
           (
              accwithdrawn . _dispensecash_(amount) . _card_ . UI
              +
              accnotenoughbalance . _notenoughmoney_ . 0
           )
           +
           notenoughcash . _atmempty_ . _card_ . UI
        )
        +
        invalidpin . _wrongpin_ . _card_ .  UI
     )




CashDispenser =
   Dispenser(100000)[cash/output, checkavailablecash/check,
                     enoughcash/enough, notenoughcash/notenough,
                     dispensecash/dispense]

Dispenser(available) =
   check(nr)
 . if available < nr then
      _notenough_ . Dispenser(available)
   else
      _enough_ . Dispenser(available)
   |
   dispense(nr) . _output_(nr) . Dispenser(available-nr)


PinCheck = checkpin(cardnr, pin)
         . if :ValidPin(cardnr, pin) then
              _validpin_ . PinCheck
           else
              _invalidpin_ . PinCheck

Accounts = accwithdraw(account, amount)
         . if :WithDrawFromAccount(account, amount) then
              _accwithdrawn_ . Accounts
           else
              _accnotenoughbalance_ . Accounts
```

Figure 5.6: ATM system example in CCS, part 2

`:Print` to print to the screen how the transaction went. If the transaction is successful the customer gets *cash*, if not he is notified of one of the error conditions, which are *wrongpin*, *notenoughmoney* or *atmempty*.

- The interface process, UI interacts with the customer as well as the Bank and CashDispenser. It accepts the inputs from the customer, first checks the cash dispenser to see if there is enough cash to pay out, then attempts to withdraw from the bank and notify the cash dispenser that it may pay out the requested amount. Like Customer, it uses non-deterministic choice to handle the possible errors. It also abstracts away from the underlying events, instead of allowing the customer to directly get a *invalidpin* event, it synchronizes with that event itself and offers the customer a *wrongpin* event instead. This implies that the mechanism that the UI interacts with could be changed but the UI could still offer the same interface to the outside.

- The Dispenser process is a generic dispenser, it takes a parameter that states how many items are available and then offers a *check* event which takes in a number and then offers either an *enough* event or a *notenough* event. It also offers a *dispense* event where it takes in the number of items to dispense, *output*'s that number and becomes the Dispenser process again, only with the number of available items reduced by the number that was just output.

- CashDispenser is a specialized form of the general Dispenser process. It uses re-labelling to indicate that the items in the dispenser are in fact cash, and re-labels the other events as *notenoughcash*, *enoughcash* and *dispensecash*. This shows how a generic process can be used as a base for a more specific one.

- Finally there are the two component processes of the bank, they are PinCheck and Accounts. They are similar in that they use .NET methods to check whether the operation they are performing is sucessful and then offer different events based on the outcome of the method call. PinCheck calls a `:ValidPin` method which takes in the pin number and card nr and returns `true` if they go together, it then offers the *validpin* event to notify the UI of this. Accounts attempts to directly call a `:WithDrawFromAccount` method to withdraw the requested amount, the method call returns `true` if the withdrawal was sucessful, otherwise `false`. It then signals this back to UI with events. These processes show how .NET integration can be used, the methods they call are defined in the class `Bank.Functions` which was imported at the top of the file, and is written in C#. For this example those methods simply return `true`, they could however check a database, call web services or do whatever else .NET programs can do before returning their result.

5.6 Summary

The CCS compiler is a fully working command line tool which supports both *pure* and *value-passing* CCS. In this chapter we looked at the CCS language itself, its syntax, semantics and value-passing extensions, and finally the implementation details of the compiler and how it interacts with the PLR. More practical aspects of the CCS compiler (command line parameters, where to download, license etc.) can be found in Appendix A and Chapter 8 describes how the compiler can be fully integrated into an integrated development environment.

Chapter 6

KLAIM Implementation

In this chapter we look at the second language implemented using the PLR, the Kernel Language for Agents Interaction and Mobility, or KLAIM. Unlike the CCS implementation, the PLR does not contain all the constructs needed for an implementation of KLAIM, so this chapter is a case study in how the PLR can be extended for use with other languages than CCS.

6.1 Overview

KLAIM is a language introduced in 1998 in [26], it's main purpose is to model mobile agents in a distributed environment. It derives many of its constructs from process algebra, but it is also heavily influenced by Linda [5, 12], a distributed computing solution from which it borrows the concept of *tuplespaces*. A tuplespace is a collection of tuples of data, these tuples can be read and removed from the tuplespace or new tuples can be added to it. Selecting tuples to read is done by means of pattern matching. In KLAIM, processes run in different *localities*, each of which contains zero or more processes and its own tuplespace. The actions performed by the processes input, output and read tuples of data from these tuplespaces. Unlike CCS, where processes synchronized with each other through channels, in KLAIM the way processes communicate is strictly by adding and retrieving data from tuple spaces of different localities.

The concept of data stores (or tuplespaces) and actions which operate on them is something that is not built into the PLR. KLAIM does however contain many constructs which are a part of the PLR, namely parallel composition, non-deterministic choice and action prefixing (albeit with different types of actions than those built into the PLR). It also contains a fairly common process algebra construct which is not part of the PLR, *replication*. These additional features make KLAIM an ideal candidate to use as a test of how easily the PLR can be extended to accommodate other languages than CCS.

6.1.1 The implemented subset of KLAIM

The implementation of KLAIM described in this chapter is only for a subset of the full language. In particular the **in**, **out** and **read** actions are implemented while the **eval** and **newloc** actions are not. The implementation also does not support process constants, processes are defined directly in the *net* and so cannot turn into other processes. The subset of KLAIM implemented here is the same as that described in [14, 2], both of which are projects developed at the Technical University of Denmark that focus on an aspect-oriented version of KLAIM. Section 6.2 only shows syntax and semantic for the subset being used, for a description of the full KLAIM language we refer to [26].

KLAIMS stated purpose is to be a programming language for programming mobile agents in a distributed environment. However in this implementation all processes in the system are running on the same computer, and in the same operating system process. The distributed part of it is purely conceptual. As such, this implementation can be seen more as a simulation of how KLAIM works rather than an implementation ready to be used for actual applications. Examples of other implementations of KLAIM are given in the next section.

6.1.2 Other implementations of KLAIM

Noteworthy implementations of KLAIM include KLAVA[4], which is a Java library representing the KLAIM constructs as Java classes, and X-Klaim [3], a compiler for a superset of KLAIM whose output is Java source code that uses the KLAVA library. A different approach is taken in [2] where a virtual machine is developed specifically for AspectK, an aspect-oriented version of KLAIM. A further explanation of these implementations and a comparison with the PLR is given in Section 9.1.

6.2 Syntax and semantics

6.2.1 Syntax

A KLAIM net is made up of located processes and located tuples. We use N for a net and the composition of nodes of the net is given with the $||$ operator. We use P and Q for processes and a for actions. x is used as a generic variable name, $!x$ represents the binding of a value to variable x and e represents an arithmetic expression. l is a locality constant while ℓ can refer to either a locality constant or a variable. t represents a tuple of elements, which can be constants, variables and variable bindings. Located tuples are written as $l :: \langle t \rangle$ and located processes as $l :: P$. Parallel composition and non-deterministic choice is represented the same way as in CCS, with $|$ and $+$ respectively. The replication construct is represented by an asterisk, $*$, immediately preceding a process. Action prefixing is written as $a.P$, and the nil process as **nil**. Papers on KLAIM traditionally allow the omission of the **nil** at the end of a process term and so does this implementation, $\mathbf{out}(t)@l.\mathbf{in}(t)@l$ is equivalent to $\mathbf{out}(t)@l.\mathbf{in}(t)@l.\mathbf{nil}$. Figure 6.1 shows an overview of the syntax.

6.2.2 Structural Congruence

Figure 6.2 shows the structural congruence of net composition, parallel composition and replication. The most interesting thing here for the implementation is that all processes located at the same locality should be treated as a parallel composition process. The replication construct also poses some interesting challenges since effectively there can be endless instances of a replicated process. Obviously this is unfeasible for implementation so some solution needs to be worked out that simulates this.

6.2.3 Semantics

Since KLAIM has actions that are quite different from those of CCS, we now look closer at those actions. As stated before the operations implemented are **in**, **out** and **read**. The reaction semantics for these operations are shown in Figure 6.3, below is an informal description with examples.

The **out** operation adds a tuple to the tuplespace of a particular locality. The operation $\mathbf{out}(\mathsf{John},\ \mathsf{Sally})@\mathsf{BusStop}$ adds the tuple $\langle \mathsf{John}, \mathsf{Sally} \rangle$ to the tuplespace

$$\begin{array}{rcll}
N & ::= & N_1 \mid\mid N_2 & \text{(Net composition)} \\
 & | & l :: \langle t \rangle & \text{(Located tuple)} \\
 & | & l :: P & \text{(Located process)} \\
 & & & \\
P & ::= & \mathbf{nil} & \text{(Nil process)} \\
 & | & a.P & \text{(Action prefixing)} \\
 & | & P \mid Q & \text{(Parallel composition)} \\
 & | & P + Q & \text{(Nondeterministic choice)} \\
 & | & *P & \text{(Replication)} \\
 & & & \\
a & ::= & \mathbf{out}(t)@\ell & \text{(Add tuple to the } \ell \text{ tuple space)} \\
 & | & \mathbf{in}(t)@\ell & \text{(Remove tuple from the } \ell \text{ tuple space)} \\
 & | & \mathbf{read}(t)@\ell & \text{(Read tuple from the } \ell \text{ tuple space)} \\
 & & & \\
t & ::= & e & \text{(Expression)} \\
 & | & \ell & \text{(Locality constant or variable)} \\
 & | & x & \text{(Variable)} \\
 & | & !x & \text{(The binding of variable x)} \\
 & | & t_1, t_2 & \text{(Sequence of tuple elements)}
\end{array}$$

Figure 6.1: KLAIM Nets and processes syntax

at the BusStop locality. The **out** operation is asynchronous and can never block. This is a big difference from the synchronization actions of CCS which are always dependent on other processes. A KLAIM process composed entirely of **out** actions can run all the way through without ever stopping because of something other processes are doing.

The **in** operation removes a tuple from a particular locality's tuplespace if it matches a certain *template*. The template is a tuple which can contain constants, variables and variable bindings. For example the operation **in**(John, !other)@BusStop will look in the BusStop locality's tuplespace for a matching tuple, and if it finds one it will be removed and a value bound to the other variable. The details of matching of templates to tuples is further explained in Section 6.2.4. One **in** operation only removes one tuple, if more than one tuple match the pattern then one of them is randomly selected. If no matching tuple exists at the tuplespace the operation will block until one becomes available.

The **read** operation is very similar to the **in** operation; essentially it behaves

$$l :: P_1 \mid P_2 \equiv l :: P_1 \mid\mid l :: P_2 \qquad\qquad l :: \ *P \equiv l :: \ P \mid \ *P$$

$$\frac{N_1 \equiv N_2}{N \mid\mid N_1 \equiv N \mid\mid N_2}$$

Figure 6.2: KLAIM Structural Congruence.

$$l_1 :: \mathbf{out}(t)@l_2.P \rightarrow l_1 :: P \mid\mid l_2 :: \langle t \rangle$$

$$l_1 :: \mathbf{in}(t)@l_2.P \mid\mid l_2 :: \langle t' \rangle \rightarrow l_1 :: P\theta \qquad \text{if } match(t; t') = \theta$$

$$l_1 :: \mathbf{read}(t)@l_2.P \mid\mid l_2 :: \langle t' \rangle \rightarrow l_1 :: P\theta \mid\mid l_2 :: \langle t' \rangle \qquad \text{if } match(t; t') = \theta$$

$$\frac{N_1 \rightarrow N_1'}{N_1 \mid\mid N_2 \rightarrow N_1' \mid\mid N_2} \qquad \frac{N \equiv N' \quad N' \rightarrow N'' \quad N'' \equiv N'''}{N \rightarrow N'''}$$

Figure 6.3: KLAIM Reaction Semantics (on closed nets).

exactly the same way, except that it does not remove a matching tuple, it just binds values to variables.

6.2.4 Pattern matching against tuples

Figure 6.4 shows the *match* function used by both the **in** and **read** operations. Essentially it works by selecting a tuple from the tuplespace that fulfills these conditions:

1. The template tuple t contains the same number of elements as the candidate tuple t' from the tuplespace.
2. In every index position i in t which is not a variable binding, the element at t_i is equal to the element at that position in t'. To put it more succinctly, $t_i = t'_i$ for all i where t_i is not a variable binding.

If a tuple does not fulfill these conditions it is not selected. If no tuple in the tuplespace fulfills the conditions then the process is blocked until the tuplespace acquires such a tuple. The result of the *match* function is a binding of variable names to values according to the variables position in the tuple. To give an example of this, suppose we have the following net:

$$match(\langle\rangle; \langle\rangle) = id$$

$$\begin{array}{ll} match(\langle t_1, \cdots, t_k\rangle; \langle t'_1, \cdots, t'_k\rangle) = & \text{let } \theta = \text{case } t_1 \text{of} \\ & \qquad\qquad \ell: \quad \text{if } t_1 = t'_1 \text{ then } id \text{ else } fail \\ & \qquad\qquad !x: \quad [t'_1/x] \\ & \text{in } \theta \circ match(\langle t_2, \cdots, t_k\rangle; \langle t'_2, \cdots, t'_k\rangle) \end{array}$$

Figure 6.4: KLAIM Pattern Matching of Templates against Tuples.

```
YellowPages::<John, 352468>
|| Sally::read(John, !phonenr)@YellowPages
```

The process at locality Sally will match the tuple of the read operation, (John, !phonenr) against the tuple ⟨John, 352468⟩ at the YellowPages locality. Since the first element matches, and the only other element is a variable binding, the tuples match and the value 352468 is bound to the variable phonenr that can then be used in the continuation of the process at locality Sally.

6.3 Implementation

The KLAIM implementation was written in C# using Visual Studio 2008 as the development environment. It implements the subset of KLAIM described in Section 6.2. KLAIM contains a number of features not available in the PLR, some of these were implemented as syntax tree nodes for the PLR syntax tree while others were implemented in a KLAIM runtime library. The PLR syntax tree nodes for action prefixing, the nil process, parallel composition and nondeterministic choice were used without modification. Expression nodes (constants, variables, arithmetic expressions) could also be re-used so right away the implementation had a number of building blocks ready for use. The main focus was then on the things that differentiate KLAIM from other process languages: its syntax, replication and tuplespaces with their associated **in**, **out** and **read** actions. Each of these is described in the following sections.

6.3.1 Replication

Replication is a construct which was quite tricky to implement according to specification. As we saw in Figure 6.2 the congruence of replication is defined as

$$l :: *P \equiv l :: P \mid *P$$

so essentially a replicated process is equal to infinite instances of that process running in parallel. For obvious reasons the implementation can not create infinite instances so another solution was needed that mimicked this behavior as closely as possibly. A number of possible approaches were considered for this problem.

The naive approach is to let the replicated process simply keep spawning new instances of itself endlessly. The way this could be implemented is by having a small delay between each instantiation of the process, to avoid using up all available memory instantly. This still has the problem of overshadowing everything else that is going on in the system. Consider the case where the first action of a replicated process is an **out** action, e.g. $\mathbf{out}(X, Y, Z)@Loc$. Over time the Loc locality will fill up with $\langle X, Y, Z\rangle$ tuples, making visualization of the net difficult as well as following what actions are happening. This could be partially alleviated by showing tuples that are identical as just one tuple with a number indicating how many of them there are, but that is still a hack to cover for an inadequate solution.

Another approach could be to change the semantics of replication to that of repetition. That is, instead of defining a replicated process as

$$l ::\ *P \equiv l ::\ P \mid *P$$

it could be defined as

$$l ::\ *P \equiv l ::\ P. *P$$

This is certainly possible to implement and is somewhat similar to replication. There are many ways in which it is different though and some hard questions are raised. What should happen to variables that are bound in P, are their values carried on into the next iteration of P, or does the next iteration start with the original values (or lack of values) of those variables? Another concern is that if P blocks on an action at any point then no new instances of P are spawned until the original instance of P continues and finishes.

The third approach would be to replace infinity with a set number. Instead of having infinite instances of P the system would instead have n instances, and

n could be configured, for instance with a command line switch. The problem with this is that it turns a behavior that is supposed to continue forever into a behavior that ends at a particular time, that is when all n instances have finished.

A fourth approach is to spawn new instances of the process only when needed so that it appears as though there is an infinite amount. The objective is to simulate infinite processes with as few processes as possible, in order to do that it is important to realize where infinite processes give the same result as a single process:

1. When a process P is blocked doing an **in** action, it does not matter whether one instance is blocked or many. The only difference is when an instance becomes unblocked, if there was just one to begin with then no instance is blocked at that action any longer, but if there were n instances blocked there are now $n - 1$ instances blocked. This implies that if there was just one instance blocked, and another instance was spawned as the first instance unblocked then that would give the same result as having infinite instances blocked.

2. Since **read** actions do not change the state of tuplespaces in any way, it can be assumed that having one instance of P perform a **read** action or having infinite instances of P perform the action would have the same effect on the state of the tuplespaces, that is, not affect them at all.

3. Mimicking the **out** operation is a little more tricky than the other two. Clearly there is a difference between one process performing an **out** action and multiple processes performing the same action. If multiple processes perform the action then there will be multiple copies of the same tuple (or rather multiple tuples with the same value) at the tuplespace that the action was performed at (ignoring possible differences in variable values). So from the perspective of looking at the tuplespace there clearly is a difference. However, if we consider how this affects the overall system, the behaviour of other processes, then the difference is only apparent when other processes start removing those tuples from the tuplespace. If only one process outputs a tuple t then the next process that tries to remove it will be successful but any subsequent process that tries to remove it will block. However if there were multiple copies of t then no process trying to remove it would block. A possible way to mimic this would be to track each tuple that is output by a replicated process, and if that tuple is removed by another process then a new instance of the replicated process can be spawned which will output a new identical tuple t at the tuplespace, allowing more processes to sucessfully perform an **in** action at that tuplespace.

The first attempt at replication tried to use the approach of simulating infinite processes by spawning new instances at certain points in the program. After quite a lot of time had been spent implementing that in a satisfactory way, a number of problems became obvious. These include:

1. When the replicated process starts with a **read** action which does not block, followed by an **out** action that uses variables bound by the previous action, then enough processes must be spawned to account for all possible variations that might have happened. Consider the following net:

   ```
      Loc::<A,1>
   || Loc::<A,2>
   || Loc::<A,3>
   || Loc::* read(A, !nr)@Loc . out(B, nr)@OtherLoc
   ```

   The selection of tuples when more than one match the template is non-deterministic, so the variable nr might be bound to either 1, 2 or 3. Were this done with infinite processes then all of the possibilities would be bound and so the end result would be that OtherLoc ended up having at least one instance of each of the tuples $\langle B, 1\rangle$, $\langle B, 2\rangle$ and $\langle B, 3\rangle$. To successfully mimic this behaviors would require the program to constantly generate all combinations of what might happen during **read** actions. This quickly gets complicated, especially when many **read** actions occur in a row.

2. When there is a non deterministic choice between two or more processes, then it would be necessary to make at least one process take each of the paths, since if there were infinite processes making that choice then surely at least one of them would take each offered choice.

3. Timing issues further complicate matters. Consider the following fragment of a net:

   ```
   Loc::* read(!x)@Loc . in(x, !y)@OtherLoc
   ```

   In the solution suggested above a new process would be spawned when the **in** action would complete. That process would start by reading x at the locality Loc. But if the previous **in** action was blocked for some amount of time then the contents of Loc might have been changed one or more times in that period. That implies that a process that was spawned after the **in** action was completed might read a different value of x at Loc than it would have if the two processes were actually started in parallel.

After considering these problems as well as some other exceptions and corner cases it became apparent that mimicking replication in a general way was not feasible. Instead it was implemented in a restricted way, by requiring that every replicated process starts with an **in** action. The replicated process itself can then only be an action prefix process, it cannot be a parallel composition or non deterministic choice between processes. It can however start by performing an **in** action and then turn into a parallel composition or non deterministic choice. The initial **in** action serves as a guard that blocks until a tuple is available, and as soon as it is unblocked it spawns a new instance of the replicated process. It is intuitively clear that this does in fact preserve the congruence of replication, if there were infinite processes then how many of them would start running would depend on the number of tuples that matched the initial **in** action, by spawning a new process whenever the first **in** action has completed the same behavior occurs.

The semantics of this restricted form of replication are then

$$*a.P \xrightarrow{a} P \mid (*a.P) \quad \text{where } a \text{ is an } \textbf{in} \text{ action}$$

which means that as soon as a has been performed a new $*a.P$ is started. A replicated process can be thought of like a server process, it waits until a tuple matching a particular template is available, and then removes it, starts the following process and keeps waiting for a new tuple that matches the template. This is much like how a server works, for instance a webserver that waits for incoming connections, when one is made it starts a worker thread to handle it and keeps listening for further connections.

Other patterns can be simulated within this restricted form, by having a locality whose sole purpose is to contain tuples that start a particular replicated process. Repetition, where one instance of a process finishes before the next instance is started could for example be modelled as

```
StartProc::<1>
|| Loc::* in(!start)@StartProc . out(X)@Y ... (other actions)
... . out(1)@StartProc
```

and starting a fixed number n of processes could be done by having n tuples in the initial net that match the initial action, for example

```
StartProc::<1> || StartProc::<2> || StartProc<3>
|| Loc::* in(!start)@StartProc ....
```

would immediately start three instance of the process since three matching tuples are available at StartLoc.

Implementing this was done by creating a new syntax node, ReplicatedProcess. That becomes one class, R, in the compiled file, and the inner process, P, starting with the **in** action, becomes another class. What R does at runtime is to create a new instance of P, give it a reference to itself and call its RunProcess method which runs it on a new thread. It then goes into an infinite loop, which has two steps:

1. Suspend the thread that it (R) is running on
2. Once it wakes up, create and start a new instance of P and go to the top of the infinite loop.

What P does is perform its initial **in** action, which might block for a while but once the action is finished it checks whether it has a reference to a replicated process, and if it does then it re-activates the thread that the replicated process is running on (which causes the replicated process to continue in its infinite loop). P then continues as normal and is not required to do anything else related to the replication.

6.3.2 Tuplespaces

The tuplespaces of KLAIM are implemented in the KLAIM runtime library. The library consists of just four classes, Net, Locality, Tuple and KLAIMException. Figure 6.5 shows a class diagram of the runtime library.

- The Net class represents the entire program, or net, being executed. It does very little itself and only contains a collection of Locality instances as well as methods to add and delete them.
- The Locality class has a name and a collection of tuples, which represents its tuplespace. The **in**, **out** and **read** actions are implemented as member methods of this class, this seemed a natural fit since this class holds the tuples. It also contains some convenience methods not available to KLAIM itself, such as methods to check in a non-blocking way if a tuple exists and retrieve a random tuple from the tuplespace.
- The Tuple class represents a tuple. As such it holds an ordered collection of items (integers or strings). It also contains the Matches method, this

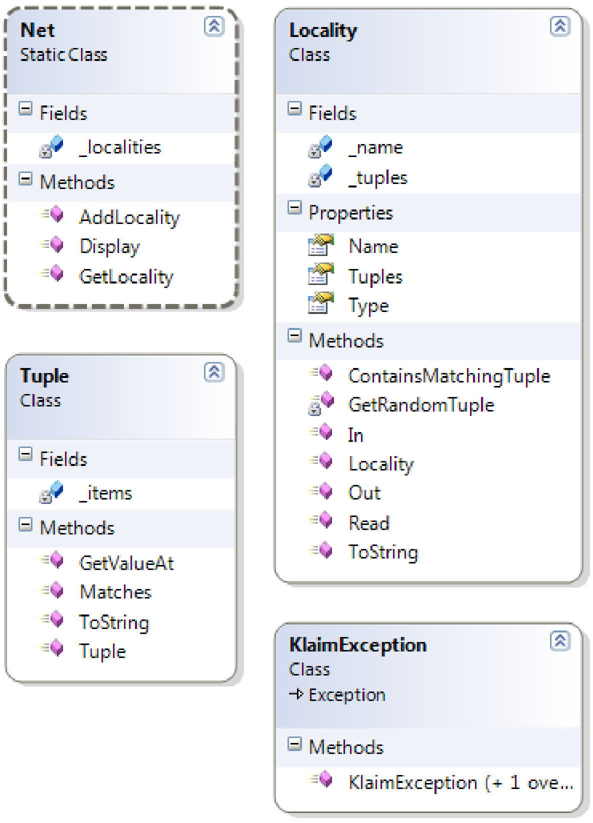


Figure 6.5: KLAIM runtime library classes

method implements the template matching against tuples described in Section 6.2.4.

- KLAIMException is a trivial class which just inherits from System.Exception and whose only purpose is to allow callers to catch a KLAIM specific exception instead of a general one.

As seen in these class descriptions, the implementation of tuplespaces is handled by these four classes, Net holds Locality instances, they hold collections of Tuple instances and the methods to access them, and the Tuple class implements the pattern matching.

The tuples initially contained in each locality's tuplespace need to somehow be put in their place before execution of the net starts. This is an activity that does not really fit into the abstract syntax tree in any particular place. To accomplish this the KLAIM compiler subscribes to the event MainMethodStart of the ProcessSystem object, which is the root node of the syntax tree. Then when the syntax tree is compiling itself and is starting to compile the entry method of the program, it raises this event and passes as an event argument the CompileContext class. It contains an initialized ILGenerator for the main method, so the KLAIM compiler can then inject its tuplespace initialization code directly into the start of the main method, before any processes are activated.

6.3.3 In, Out and Read actions

As we saw in the previous section, the runtime methods for these actions are members of the Locality class in the runtime library. However, that is only part of their implementation, before they can be used someone has to generate the bytecode to call these methods at the appropriate places. For that purpose there are three new syntax tree nodes, InAction, OutAction and ReadAction. These inherit from the Action node from the PLR, and so can be used as children of the PLR's ActionPrefix syntax tree node. These three nodes compile themselves into bytecode that gets the correct locality from the net, and then calls the correct method on that locality. These new action have Expression nodes as children which represent the items in the tuple. One new expression node was added, VariableBinding, other nodes needed were already a part of the PLR.

There is one special case for the **out** action. That is if the locality being output at is named Screen. In that case the tuple is not stored anywhere, its values are just printed to the screen. For example the action

```
out(ResultOfExpression, 3+5)@Screen
```

would produce the output

```
ResultOfExpression, 8
```

on the users screen.

6.3.4 Class overview

Here we quickly go over the main classes used in the KLAIM compiler itself.

```
LOCALITY   = [A-Z][A-Za-z0-9]*
VARIABLE   = [a-z][A-Za-z0-9]*
VARBINDING = ![a-z][A-Za-z0-9]*
NUMBER     = [0-9]+
```

Figure 6.6: Terminals of KLAIM scanner

- Program is the main class which parses the command line, validates the given parameters and instantiates the parser. It then gets the syntax tree back from the parser (given that no syntax errors occurred) and instructs the syntax tree to compile itself.
- Scanner is the lexer class whose responsibility it is to tokenize a KLAIM source code file into valid KLAIM terminals. Figure 6.6 shows the more complicated terminals of KLAIM defined by regular expressions. The simpler terminals, who are just string constants, are given directly in quotes in the parser definition.
- Parser is a recursive-descent parser for KLAIM. The parser constructs a PLR abstract syntax tree as it parses, using both the built in PLR syntax tree nodes and the additional KLAIM syntax tree nodes described below. As with the CCS parser and lexer, this class as well as the Scanner class are generated with help from the Coco/R [25] parser generator. The Extended Backus-Naur Form (EBNF) description of the full syntax, which is used as input to Coco/R, is shown in Figure 6.7.
- InAction, OutAction and ReadAction are syntax tree nodes for the **in**, **out** and **read** operations, respectively. They are stored in the compiler rather than the runtime library since they are only needed at compile time.
- VariableBinding is a syntax tree node which inherits from Expression and represents the binding of a value from a tuple to a variable.

6.4 KLAIM invoice system example

The preceding sections in this chapter have only presented small snippets of KLAIM code. To get a clearer view of what an actual system modeled in KLAIM might look like, an invoice system example is shown in Figure 6.8, and explained further below. We begin by looking at the localities in the system:

- Inbox is a locality which contains invoices that have just arrived at the company which the net represents. The Inbox contains no processes, it

```
KLAIM =
  LocatedItem { "||" LocatedItem }

LocatedItem =
  LOCALITY "::" (Tuple | Process)

Tuple =
  "<" Constant { "," Constant } ">"

Constant =
  LOCALITY | NUMBER

Process =
  ["*"] NonDeterministicChoice

NonDeterministicChoice =
  ParallelComposition { "+" ParallelComposition }

ParallelComposition =
  ActionPrefix { "|" ActionPrefix }

ActionPrefix =
  Action [ "." ActionPrefix
  | "(" Process ")"
  | "nil"

Action =
  OutAction | InOrReadAction

OutAction =
  "out"
  "(" OutParam { "," OutParam } ")"
  "@" (LOCALITY | VARIABLE)

OutParam =
  LOCALITY | Expression

InOrReadAction =
  ("in"|"read")
  "(" InOrReadParam {"," InOrReadParam } ")"
  "@" (LOCALITY | VARIABLE)

InOrReadParam =
  LOCALITY | VARBINDING | Expression

Expression =
  PlusMinusTerm { ("+"|"-") PlusMinusTerm }

PlusMinusTerm =
  UnaryMinusTerm { ("*"|"/"|"\%") UnaryMinusTerm }

UnaryMinusTerm =
  ["-"]
  (
    "(" Expression ")"
    |
    NUMBER
    |
    VARIABLE
  )
```

Figure 6.7: EBNF Productions of KLAIM

```
   Inbox :: <Incoming, Paper, OfficeSupplies, 200, John>
|| Inbox :: <Incoming, Printer, OfficeSupplies, 500, John>
|| Inbox :: <Incoming, Textbooks, BookShack, 100, Alice>

|| Ledger :: <Paid, Cake, 50, AlfredosBakery, Alice>

|| Budget :: <John, 2000>
|| Budget :: <Alice, 5000>

|| Secretary :: * in(Incoming, !item, !vendor, !amount,
                     !person)@Inbox
               . out(Received, item, vendor, amount)@person

|| Secretary :: * in(Denied, !item, !vendor, !amount, !person)@self
               . out(Denied, item, vendor, amount)@vendor

|| John :: * in(Received, !item, !vendor, !amount)@self
          .
          (
          out(Confirmed, item, vendor, amount, John)@Finance
          +
          out(Denied, item, vendor, amount, John)@Secretary
          )

|| Alice :: * in(Received, !item, !vendor, !amount)@self
          .
          (
          out(Confirmed, item, vendor, amount, Alice)@Finance
          +
          out(Denied, item, vendor, amount, Alice)@Secretary
          )

|| Finance :: * in(Confirmed, !item, !vendor, !amount, !person)@self
             . in(person, !budget)@Budget
             . out(person, budget - amount)@Budget
             . out(Paid, item, vendor, amount, John)@Ledger
```

Figure 6.8: Invoice system example in KLAIM

is just a database for invoices. The invoices themselves are tuples which contain five elements: a status tag, the item being ordered, the vendor name, the price, and the name of the person who ordered the item. For example the tuple $\langle Incoming, Printer, OfficeSupplies, 500, John\rangle$ is an invoice that has the status Incoming, is for a printer that was ordered by John from OfficeSupplies and cost 500.

- Ledger represents the general ledger of the company. Invoices end up there after they have been confirmed and paid. Every invoice in the ledger should have its status as Paid.
- The Secretary locality represents the company secretary who removes invoices from the Inbox and then sends them on to the person who ordered the item in question, with a status tag of Received. Note here that the process is replicated and starts with an **in** action, whenever an invoice (tuple) is available at Inbox whose first element is the constant Incoming a new instance of the Secretary process will be spawned. Also note how the **in** action binds the variable *person* and then uses that as the locality in the following action **out**(...)@person. Secretary also contains another process running in parallel, this process reads in invoices with the status Denied and then sends them back to the vendor that issued the invoice.
- The localities John and Alice are both employees of the company and have the same process for receiving invoices. They are replicated like the Secretary process so they start whenever a matching tuple lands in their tuplespace. They read in the invoice and then have a choice where they can either send it back to the secretary with the status Denied or send it on to the finance department with the status Confirmed.
- Budget is a database of how much money each employee may spend on outside purchases. It has entries for John and Alice and their current budgets.
- Finally, Finance is the finance department of the company. It inputs Confirmed invoices from its tuplespace, then subtracts the amount spent from the budget of the person who ordered the item. It does this by first removing the appropriate tuple from Budget by matching on the name on the invoice. It then outputs a new tuple with the current budget which is a result of an evaluated expression, $budget - amount$. The process then ends by adding the invoice to the Ledger with a status of Paid.

This example system shows how KLAIM can be used to model real world systems. Its high level of abstraction makes it useful for modeling for example workflows without thinking about concrete implementations and technical issues. Localities can be used as simply databases, such as the Budget, as actors, such as Secretary, or as both. This is for example the case with John and Alice, they have processes that they follow, and those processes operate on data in the self tuplespace which implies that they own the data. At runtime self simply resolves to the name of the locality where the process is running.

6.5 Summary

The KLAIM compiler was mainly meant as a proof-of-concept that the PLR could in fact be used and be useful for languages other than CCS. It serves as a case study of how a new language can be implemented fairly quickly by using the PLR as a foundation and adding compile time and/or runtime classes as needed. Although the compiler does not implement the full KLAIM language, it does implement a useful subset of it and could be used as a foundation for adding extensions to, such as implementing AspectK [14]. Another possible extension would be to add the possibility of calling .NET methods, this was not done here as it had already been shown possible in the CCS implementation.

Practical aspects of the compiler (command line parameters, where to download, license etc.) can be found in Appendix A.

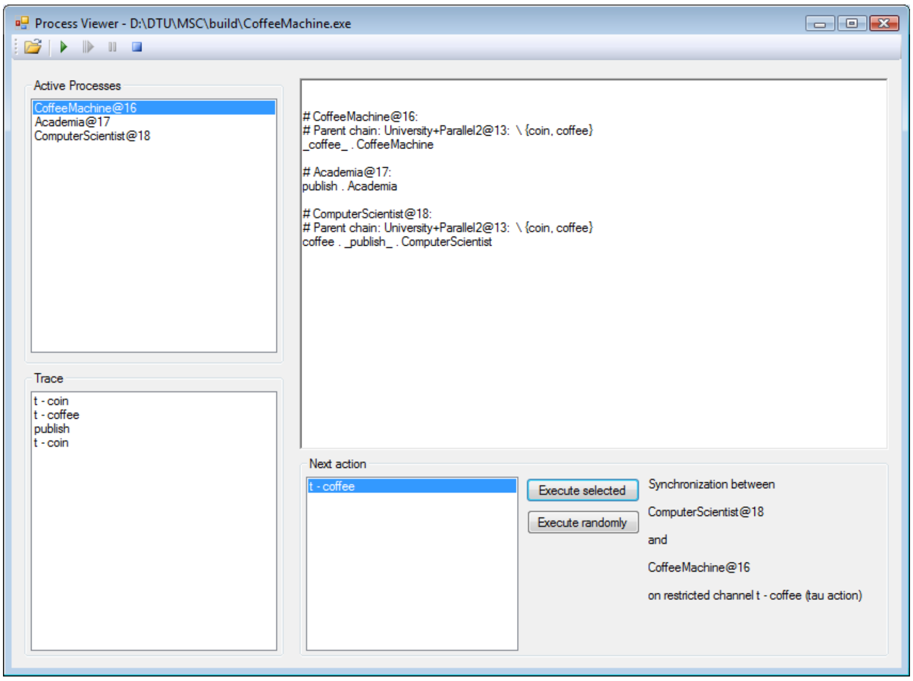
Chapter 7

Interactive Process Viewer

In this chapter we look briefly at *Process Viewer*, a tool to interact with process language applications during execution. Its architecture is explained, as well as how it interacts with compiled process language executables. The challenges in making it general enough for any process language are also discussed.

7.1 Overview

Process Viewer (hereafter referred to simply as *the viewer*) is an application to allow users to closely monitor and affect how compiled process language applications are executed. It enables the user to see what processes are currently active, what actions have been executed and what actions are ready for execution. It can be run interactively, which allows the user to select the next action for execution. If the source code for the application being run is available, then it is also possible to see the state of the system in source code form at every stage (e.g. if the original system contained the process $a\ .\ b\ .\ P$ and action a has been executed then the system now contains the process $b\ .\ P$). The user interface is simple and consists of only one window. Figure 7.1 shows a screenshot of the program during execution of a process language application. The list of active processes is in the upper left of the screen, the trace is on the lower left. The current state of the system is shown in the large text box and under it the next possible actions are shown, as well as some controls to select the action.



```
CoffeeMachine@16
Academia@17
ComputerScientist@18
```

```
# CoffeeMachine@16:
# Parent chain: University+Parallel2@13: \ {coin, coffee}
_coffee_ . CoffeeMachine

# Academia@17:
publish . Academia

# ComputerScientist@18:
# Parent chain: University+Parallel2@13: \ {coin, coffee}
coffee . _publish_ . ComputerScientist
```

```
t - coin
t - coffee
publish
t - coin
```

```
t - coffee
```

```
Synchronization between
ComputerScientist@18
and
CoffeeMachine@16
on restricted channel t - coffee (tau action)
```

Figure 7.1: The Process Viewer application

7.2 Architecture

7.2.1 Class structure

The application is written in C# and is made up of just two main classes. ProcessViewer is the class for the window itself and contains all the actions that have to do with the graphical user interface. The other class is ProcessStateVisualization which is responsible for keeping track of how the system looks in source code form at every stage of the execution. Since most of what is done in the program has to to with updating controls in the window it was not deemed necessary to modularize the code further. The only real algorithm in the program is how the process state is extracted from the running process language application, that code was clearly not tied to the graphical user interface and therefore it was put in its own class, the ProcessStateVisualization class.

7.2.2 Interaction with the PLR

The process language application that is being executed is run in the same operating system process as the process viewer itself. This is done by loading the process language application assembly and simply calling its Main method on a seperate thread. Doing it this way allows the viewer to interact directly with the process language application and its classes. The interaction happens mainly through the PLR's Scheduler class. The scheduler has three useful events that the process viewer subscribes to, ProcessRegistered, ProcessKilled and TraceItemAdded. These events notify subscribers when new processes are added to the system, when processes are removed from the system, and when new items are added to the trace, that is when actions have been executed. The process viewer uses these events to update the controls that show active processes and the trace.

To allow the user to select which action to execute the viewer makes use of a simple abstraction that the Scheduler class provides. The scheduler does not have a special method to select an action from all the candidate actions, instead it has a delegate (function pointer) to a method that takes in a list of CandidateAction classes and returns the one that should be executed. This function pointer by default points to a simple method that randomly chooses an action to execute, but it can be set to any other method that has the correct method signature. The viewer sets this function pointer to its own method, that method shows each candidate action in the window and returns the one the user chooses. The CandidateAction class contains information about an action and the process or processes that perform it so the viewer has enough data to display to the user.

7.2.3 Process visualization

Showing the state of the process system after each step requires the original source file. This is optional, if the source file is not available then the viewer can still be used, but the system state in source code form will not be shown. The names of the active processes, the trace and the candidate actions all work with just the compiled executable though. The reason for this is that those things can all be shown in a reasonable way using the class names and `.ToString()` methods of the runtime classes whereas displaying the system in source code form requires the abstract syntax tree, which is no longer available after the application has been compiled.

This is further complicated by the fact that the viewer is not specific to a particular process language, it is meant to work with any language that uses the PLR. The problem then becomes how can the viewer know which parser to use to parse the abstract syntax tree from the provided source file. To solve this the viewer has an associated configuration file which lists the filenames of all assemblies that contain parsers. The parsers can then implement an IParser interface that is provided by the PLR. The interface has a Parse method as well as properties for the language name and the file extensions used by that language. The viewer inspects the assemblies listed in the configuration file and loads all classes that implement IParser. When a source file is selected the viewer can then lookup the correct parser for it by filename extension and use that parser to parse the file, or throw an error if no suitable parser is found. The IParser interface also contains a property that returns a BaseFormatter instance (previously discussed in Section 3.3.2) which can be used to get a source code representation of the whole, or parts of, the abstract syntax tree.

Once the abstract syntax tree and an appropriate formatter for it are loaded then the ProcessStateVisualization class can create a text representation of the system by starting with the initial processes and then keeping track of which actions are performed and removing the corresponding nodes from the syntax tree. The formatter is then used on those parts of the syntax tree that are active at a given time and the source code for those active processes is shown in the large textbox in the main Process Viewer window.

The text representation of the system shown in the window is not exactly like the original source code. Below is an example of how a process is displayed:

```
# CoffeeMachine@15:
# Parent chain: University+Parallel2@14:  \ {coin, coffee}
coin . _coffee_ . CoffeeMachine
```

The first line shows that this is a process named CoffeeMachine and it has an id of 15. The id is the thread id of the thread that the process runs on. The next line shows the process's parent chain, that is which process spawned it. In this case we see that it was spawned by the second parallel branch of the University process, and that the restrictions that apply are that coin and coffee are hidden. If the parent chain were longer then additional ancestors would be shown on the same line. Finally the third line shows the process as it looks now, $coin\ .\ \overline{coffee}\ .$ CoffeeMachine. Once the $coin$ action has been performed this will change to read just $\overline{coffee}\ .$ CoffeeMachine.

There are a few limitations to this text representation. One is that it cannot show process systems that contain `if-then-else` constructs. This is because the program cannot figure out from the compiled executable which path has been taken in the conditional construct and it cannot evaluate the `if` condition itself. The text representation also does not show non-deterministic choice and parallel composition as they normally appear in source code form, instead it just shows each of the process branches separately under the class names that they are given in the executable, e.g. A+NonDeterministic1 and A+NonDeterministic2 are two branches in the same choice, even though they are not shown together as one process.

7.3 Summary

The Process Viewer application is a useful tool for monitoring and interacting with compiled process language applications. Due to the interoperability of .NET the application can directly interact with running process language applications and show processes, traces and candidate actions. It also allows the user to select which actions are executed in the process language application and shows the state of the processes as they change. When creating and working with process language applications the Process Viewer really helps the user to understand and follow what is happening during execution.

Chapter 8

Integrated Development Environment

One of the biggest differences between academic programming languages and industrial programming languages is the level of tool support. Academic languages traditionally are edited in text editors and compiled with command line tools, while industrial languages usually have integrated development environments (IDE's). These environments offer a wide range of features such as syntax highlighting, instant visual warnings about syntax errors, background compilation, automatic listing of available methods and variables (IntelliSense), built in debuggers and refactoring. One of the goals of this project was to explore how well the CCS language could be integrated into one of these environments and how it could benefit from the features they have to offer. This chapter presents the results of this exploration.

8.1 Choice of Integrated Development Environment

When choosing which IDE would be most suitable for CCS two environments stood out, Microsoft Visual Studio 2008 and SharpDevelop. Eclipse was briefly considered as well since it provides good plugin support, but was dismissed since it is built on Java and since this project is focused on integrating with the .NET framework it seemed natural to go with a .NET development environment. Below the two candidate environments are described and reasons given for choosing one of them.

8.1.1 Microsoft Visual Studio 2008

Microsoft's Visual Studio is the most popular environment for .NET development. Visual Studio versions are released alongside new versions of the .NET framework itself and each new version takes full advantage of and supports all the new features in the .NET framework. The latest version as of this writing is Visual Studio 2008 which supports the .NET framework 3.5. Visual Studio has an extensive extensibility API based on COM technology, an older technology for interaction between programs written in different programming languages. Languages are integrated by creating *language services*, the Visual Studio program itself is simply a host for these services. The languages that come with the .NET Framework, C# and Visual Basic.NET have their own language services that do not have any special access to Visual Studio, this implies that a language service for a new language can be made to offer all the same features as those supported by the built in languages. The downside of Visual Studio is that its extensibility API is fairly complicated and hard to work with. Another drawback is that even though there exist a fair number of samples for language services, the professional level services for languages like C# and Visual Basic.NET are not available as open source so it is not possible to look at them for inspiration.

8.1.2 SharpDevelop

The second candidate for a CCS development environment was SharpDevelop, an open source IDE written entirely in C#. SharpDevelop is very similar to Visual Studio in look and feel, and offers many of the same features. Its extensibility API is entirely in .NET and is in many ways cleaner and clearer than Visual Studio's API. It also has the benefit of being open source software, so it is easy to get a clearer picture of its architecture, and view the source for other language services, even the built in ones for C# and Visual Basic.NET. The architecture of SharpDevelop itself has even been the subject of a book, [17]. The drawbacks are that its debugger is inferior to Visual Studio's, the application itself is slower, and it is not as well known as Visual Studio. It also does not offer all the same features as Visual Studio, a notable feature that is missing is the ability to highlight syntax errors as the user types in code.

8.1.3 Chosen environment

After researching both environments, Visual Studio 2008 was chosen as the one to implement CCS's language service in. This was based primarily on the fact that Visual Studio is the IDE of choice for most .NET developers, it is fast, offers great debugging support and real time syntax checking. While the standard versions of Visual Studio are not free, it is possible to download only the Visual Studio shell and distribute it for free. The shell is the Visual Studio program itself without any language services. This makes it possible to offer the CCS development environment free of charge to anyone who wishes to use it.

8.2 Building a language service

8.2.1 Goal

The goal of this integration was to be able to use Visual Studio to manage all aspects of working with the CCS language. To achieve that the following features needed to be implemented:

1. **CCS Projects**. The ability to create new projects specifically for CCS applications.
2. **Syntax highlighting**. To have different tokens of the language colored differently so that it is easier to see and understand the structure of the code.
3. **Real time syntax checking**. Display visual warnings about incorrect syntax in the code as it is being written.
4. **IntelliSense**. Allow the developer to press a keyboard shortcut and get a list of all available channel names, keywords, variable names and process names in the current application, and insert them at the current location in code.
5. **Match braces**. When working with large expressions it can be hard to see which braces (parentheses, curly braces, angle brackets) match, and missing parentheses are a common syntax error. Visual Studio can highlight the matching braces automatically, if the language service provides it with the necessary information about which braces match each other.

6. **Build support**. To be able to compile the code being written from within Visual Studio, using its *Build* menu items and commands. A part of that is being able to use Visual Studio's built in mechanism to search for and add references to other .NET assemblies that the application uses, and pass those references to the compiler at compile time.

7. **Debugger support**. Launching the application after it had been built and attaching the Visual Studio debugger to the running executable. Also the ability to set breakpoints and step through the code as it is executing.

Visual Studio offers the front-end for all these features, that is the graphical user interface to display them and the infrastructure that calls into the language service's code to get the data necessary for the features to work. However, Visual Studio of course has no knowledge of specific languages and so it is the responsibility of each language service to provide the back-end, the code that understands the language, its tokens, its syntax, which items to display in IntelliSense and so on. The implementation of these features is described in the following sections.

8.2.2 Visual Studio API

The COM extensibility API for Visual Studio is fairly complicated and unfriendly to use. It is also poorly documented. Fortunately Microsoft has recently released a framework called the Managed Package Framework, or MPF for short. This comes as part of the Visual Studio SDK (Software Development Kit) and is a collection of .NET classes that wrap a lot of the underlying COM interface, making it easier to work with in .NET languages. The MPF classes implement much of the tedious boilerplate code which is necessary and common to all language services. Parts of the Managed Package Framework are released only as source code and are meant to be included directly in language service projects when they are being built. In the source code repository for this project, these files have been marked specifically with a header stating that they are supplied by Microsoft to avoid confusion about which code is original work and which code is borrowed.

8.2.3 CCS Projects

To allow the user to create a new project of type CCS project four main things needed to be implemented.

1. A class named CCSProjectPackage, this class inherits from a ProjectPackage that is provided in the Managed Package Framework. It provides information about the project file ending, the project name and a path to the project templates described in item 4 in this list. It doesn't contain any real code, it only provides the necessary properties for Visual Studio to recognize that a new type of project has been registered. To ensure uniqueness of the package it has a globally unique identifier (GUID).

2. A class named CCSProjectFactory, this class inherits from a ProjectFactory class from the Managed Package Framework. It also contains a globally unique identifier and overrides only one method, CreateProject() which returns a ProjectNode instance.

3. A class named CCSProjectNode. This class represents the project once it has been created, it inherits from ProjectNode, again from Managed Package Framework. In this class it is possible to override a lot of behavior, such as what happens when build dependencies are added to the project, which items can be deleted from the project, how to clean the project and many more. This implementation did not require a lot of overrides, since each project only contains one source file so most project possibilities are simply not enabled.

4. Templates for the project needed to be created. The main template is for the project file. The project file defines which items are included and which MSBuild targets (see Section 8.2.8) should be used to build the project. Another template is for a default CCS source file that is included in every project, and contains some simple sample code to get people started. Finally there is a file named *CCS Project.vstemplate*, this contains some metadata about the templates and is the file used by Visual Studio to determine which items to show when new projects are created.

In addition to these items that needed to be implemented, a lot of source code from the Managed Package Framework is necessary to build the project package successfully. These are standard implementations of a number of interfaces Visual Studio requires, they can be overridden for more complex projects than the CCS projects. It was surprising how much source code is needed to do a relatively simple thing like creating a new project type.

8.2.4 Syntax Highlighting

Syntax highlighting is when different tokens in a language are given different color to help differentiate them when looking at code. This is very helpful when

```
public interface IScanner {
  bool ScanTokenAndProvideInfoAboutIt(
             TokenInfo tokenInfo, ref int state);
  void SetSource(string source, int offset);
}
```

Figure 8.1: IScanner interface

looking at code to quickly sense the structure and identify problems, and has been a standard feature of development environments as well as most advanced text editors for many years. In many common text editors this is simply implemented as lists of tokens and colors for them. The Managed Package Framework however requires that the language service provides an implementation of an interface named IScanner, shown in Figure 8.1.

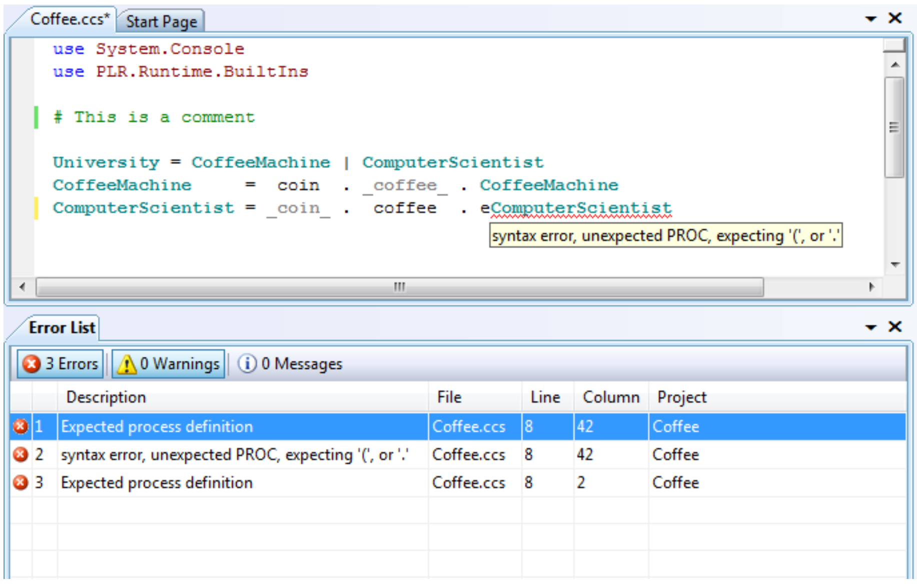
The SetSource method of the interface is called by Visual Studio and provides the IScanner with one line of source code at a time, Visual Studio then repeatedly calls ScanTokenAndProvideInfoAboutIt to get information about each of the tokens in that line. This is done on a line-by-line basis so that only lines that change need to be re-colored, as it is a relatively expensive operation. For this project the IScanner interface was implemented using a lexer class, how that class was generated is described further in Section 8.2.5. The tokens of the language were divided up into eight distinct color classes and colored as follows:

- Process constants - Greenblue
- Output actions on channels - Dark gray
- Method calls - Magenta
- Comments - Dark green
- Strings and class names - Maroon
- Keywords - Blue
- Numbers - Red
- All other tokens - Black

An example of the syntax highlighting can be seen in Figure 8.2.

8.2.5 Real time syntax checking

A very useful feature of Visual Studio is its ability to show the user errors in their code in real time, as they are typing. These errors (or warnings) are shown both as text error messages in an error message window, as well as red curvy lines under the places in code where the syntax errors occurs. This makes it extremely easy to look at a page of code and determine whether it is syntactically correct. Figure 8.2 shows an example of this feature in action.



```
use System.Console
use PLR.Runtime.BuiltIns

# This is a comment

University = CoffeeMachine | ComputerScientist
CoffeeMachine      =  coin  . _coffee_ . CoffeeMachine
ComputerScientist = _coin_  .  coffee  . eComputerScientist
                                  syntax error, unexpected PROC, expecting '(', or '.'
```

3 Errors | 0 Warnings | 0 Messages

	Description	File	Line	Column	Project
1	Expected process definition	Coffee.ccs	8	42	Coffee
2	syntax error, unexpected PROC, expecting '(', or '.'	Coffee.ccs	8	42	Coffee
3	Expected process definition	Coffee.ccs	8	2	Coffee

Figure 8.2: Syntax checking for CCS in Visual Studio

The component in Visual Studio that is responsible for syntax highlighting and syntax checking is named Babel [8]. As part of the Managed Package Framework there is a collection of classes to wrap this component, these are called the Managed Babel System. With the Managed Babel System come two programs called MPLex.exe and MPPG.exe, these acronyms stand for *Managed Package Lex* and *Managed Package Parser Generator*. These are .NET implementations of the well known parser generator tools Lex and YACC and derive directly from the *Garden Point Parser Generator* [13] developed at the Queensland University of Technology. These tools take as input a lexer.lex file and parser.y file. The lexer file defines the tokens of the language with regular expressions, and the parser file describes the syntax of the language in extended Backus-Naur form, or EBNF. From these input files MPLex.exe and MPPG.exe generate C# code for a lexer and a parser for the language. A more detailed explanation of Lex and YACC-like tools is outside the scope of this paper but a useful book on the subject is [18].

Once the generated lexer and parser have been built, Visual Studio is responsible for calling them repeatedly in the background while the user is typing code. Visual Studio calls a method named ParseSource(ParseRequest req) which is a method of the CCSLanguage class. That class inherits from BabelLanguage-Service and overrides its ParseSource method. Inside this method the parser and lexer are instantiated and parse the source code which is a part of the ParseRequest instance passed to the method. The parser then logs every error it encounters, with file name and line numbers, and Visual Studio is responsible for displaying these errors to the user.

As we saw in Chapter 5 the parser used by the CCS compiler itself was written using the Coco/R parser generator. It would have been preferable to re-use that parser directly instead of defining a new parser for the same input language. While it would have been possible, the fact is that MPLex.exe and MPPG.exe are optimized for generating parsers that work well with the Managed Babel System, and the Managed Babel infrastructure expects parsers and lexers that conform to a certain interface. For that reason MPLex and MPPG were used to create a new parser and lexer instead of re-using the existing ones. The drawback to this is of course that two implementations of the same language need to be maintained and kept in sync. However, the input language for both these parser generators is based on EBNF syntax and so it is fairly trivial to port from one to the other. In hindsight the best approach would have been to use MPPG and MPLex for the CCS compiler as well as the language service.

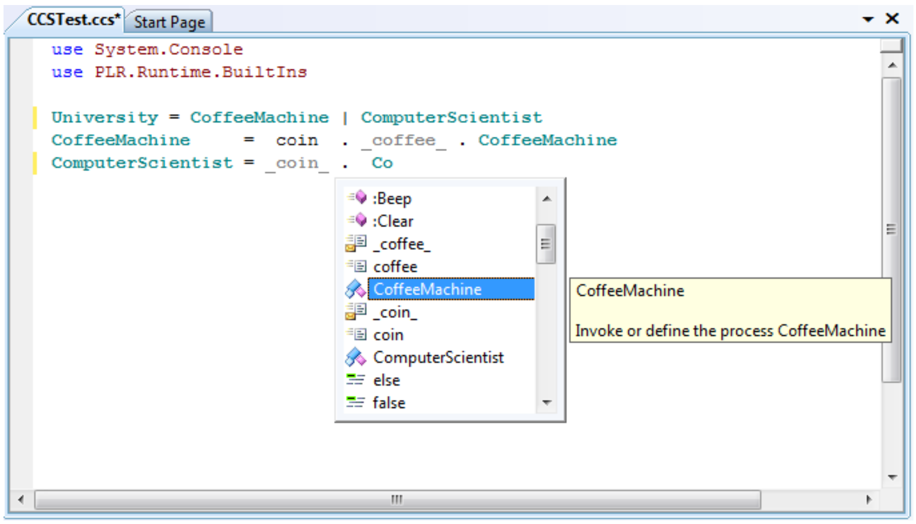
8.2.6 IntelliSense

IntelliSense (or automatic word completion) is one of the most useful features of Visual Studio. When the programmer presses the keyboard combination CTRL+SPACE at some point in the source code, the environment shows a list of items that the programmer might wish to insert at that point, and a description of them. This includes (in CCS's case) channel names, process names, imported .NET method names and CCS keywords. If the programmer invokes IntelliSense once the caret is positioned after a half completed word, such as *cof* and there is only one candidate that starts with *cof*, namely the channel *coffee*, then the word is completed automatically and the list of possibilities is not even shown.

This behaviour is implemented through two main classes, AuthoringScope and Resolver. AuthoringScope is the class that is returned from the ParseSource method we learned about in Section 8.2.5, part of its interface is the method Get-

Declarations. This method is called when the user has pressed CTRL+SPACE and it in turns calls a method named FindCompletions on the Resolver class, which returns a list of all the items to display to the user, along with descriptions and icons.

To create the list of items the resolver uses the generated scanner class we discussed previously and scans all tokens in the source file to find process and channel names. For every channel name it adds an item for the output action and input action on that channel, e.g. both `coffee` and `_coffee_`. The resolver also adds all the language keywords (if,then,use etc.) to the list so they can be completed automatically, and to provide a reference for the user about what is possible to do. Finally the resolver tries to find all methods that the application could call, and add them along with descriptions that include parameter names and types. To do this the resolver tries to look up all classes that are imported in the source code using the `use` keyword, as well as look for methods in the class PLR.Runtime.BuiltIns which contains methods that can be called without a `use` statement. The method names and parameters are found through reflection. Since the PLR only supports static methods and only strings and integers as parameters, the resolver only considers methods that fulfill that criteria. Once this has been done the list of names, keywords and methods is returned so that Visual Studio can display them to the user. Figure 8.3 shows an example of this feature in action. Note how different types of items have different icons to distinguish them from each other.



```
use System.Console
use PLR.Runtime.BuiltIns

University = CoffeeMachine | ComputerScientist
CoffeeMachine     = coin  . _coffee_ . CoffeeMachine
ComputerScientist = _coin_  .  Co
```

CoffeeMachine

Invoke or define the process CoffeeMachine

Figure 8.3: IntelliSense for CCS in Visual Studio

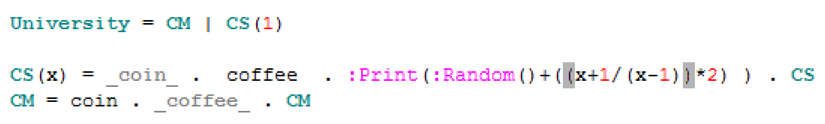
```
UnaryMinusTerm
    : '-' UnaryMinusTerm
    | NUMBER
    | '(' Expr ')'  { Match(@1, @3); }
    | LCASEIDENT
    | KWTRUE
    | MethodCall
    | KWFALSE
    ;
```

Figure 8.4: Matching braces for expressions

8.2.7 Match braces

This feature was fairly trivial to implement as most of it is provided by the Managed Package Framework. To get it working the generated parser needs to store information about every matching parentheses pair while it is parsing the source. This is simply done by calling a Match method in the parser definition. Figure 8.4 shows how the brace matching for expressions is achieved in the parser definition.

The macros @1 and @3 are converted when the parser is generated and mean that the first and third token on the lines match. The only additional thing needed to get the feature working was to check in the ParseSource method if the reason for the parsing was to highlight braces, and if so then use the braces found by the parser earlier and call a MatchPair method on the ParseRequest object that was passed to the method. Figure 8.5 shows how this feature is useful when working with large expressions.



```
University = CM | CS(1)

CS(x) = _coin_ .  coffee  . :Print(:Random()+((x+1/(x-1))*2) ) . CS
CM = coin . _coffee_ . CM
```

Figure 8.5: Brace matching for CCS in Visual Studio

8.2.8 Build support

Visual Studio has menu items and keyboard shortcuts for a *Build* command. This command is used to build (compile) the source files of the current project into an executable file. The underlying system that does the actual building is named MSBuild, and it is a command line build tool, similar to *make* which is commonly used on Unix and Linux platforms, and *NAnt*, an popular open source build tool for the .NET framework. The input files for MSBuild are XML files that define what the tool should do. The three most important elements in these files are *targets*, *tasks* and *properties*.

A *target* is a particular action to take, for instance there can be a *build* target which builds an entire project and a *clean* target which deletes all intermediate files. Targets can depend on each other, for example a *rebuild* target can depend on the *clean* and *build* targets, so that whenever *rebuild* is executed the tasks it depends on are automatically executed first.

Tasks are the operations performed by the targets. Each task is usually a single distinct action, for example one task might be to call a compiler, another task might be to copy files. Tasks can take parameters, for instance the names of the files to compile. Custom tasks can be written in .NET to achieve any operation and integrate it into the build process.

Properties are essentially variables to use in the build process, and are often passed as parameters to the tasks. A property might for instance be named *Debug* and have either the value `true` or `false`. It could then be passed to a task as a parameter.

Each language in Visual Studio has its own *.targets* files which defines all the targets and tasks specific to that language. Additionally there is a common targets file, *Microsoft.Common.targets* which all languages use. These *.targets* files are then referenced in the project files for the languages. To get CCS working with the build system it was necessary to create one custom task, to call the compiler. The task is defined in a class named CompileTask which inherits from MSBuild's ToolTask class. It has an Execute method which calls the CCS compiler and logs all errors that the compiler emits, and four properties that can be set, Debug, InputFile, OutputFile and References. To make that task available in Visual Studio the template CCS project file references a file called CCS.targets. That file in its entirety is shown in Figure 8.6.

All that is necessary to do in the *CCS.targets* file is to create a target named *CoreCompile*, and in that target call the custom compile task that was created for the CCS compiler. Empty implementations of the targets *CreateManifestResourceNames*, *Build* and *Compile* are also provided, since otherwise these targets try to perform actions that are not necessary for building CCS applications. The *Debug* parameter can be set within Visual Studio and the names of the input and output files are determined by the name given to the project when it is created. References to other .NET assemblies can be added in Visual Studio and they will be passed to the compile task through the *References* parameter. Having the CCS project file reference the CCS.targets file, and that file call

```
<?xml version="1.0" encoding="utf-8" ?>
<Project DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <UsingTask TaskName="CCS.BuildTasks.CompileTask"
       AssemblyFile="$(CCS_PATH)CCS.BuildTasks.dll"/>
    <Target Name="CoreCompile">
         <CompileTask
              Debug="$(DebugSymbols)"
              OutputFile="@(IntermediateAssembly)"
              InputFile="@(Compile)"
              References="@(ReferencePath)"/>
    </Target>
    <Target Name="CreateManifestResourceNames"></Target>
    <Target Name="Build"></Target>
    <Target Name="Compile"></Target>
    <Import Project="$(MSBuildBinPath)\Microsoft.Common.targets" />
</Project>
```

Figure 8.6: The CCS.targets file

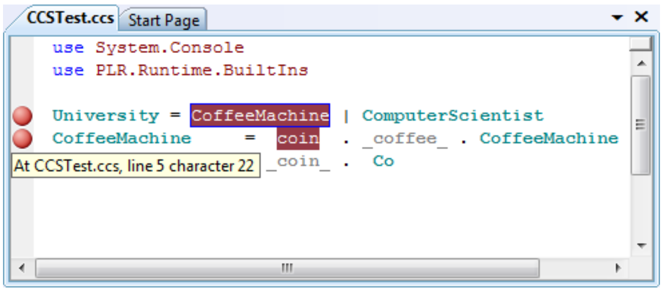
,

the custom compile task is enough to get full build support from within Visual Studio.

8.2.9 Debugger support

Once the build support described in Section 8.2.8 is in place, getting debugger support within Visual Studio is trivial. All that is needed is to select the *Debug* configuration inside Visual Studio, this passes the paramater *Debug=true* to the compile task and on to the compiler. We already saw how the PLR supports emitting debugging symbols in Section 3.3.5. Pressing the F5 key, or the play button, in Visual Studio will then build the project, launch the executable if the build is successful and attach the debugger. This works due to the fact that the *Microsoft.Common.targets* file defines a *Run* target that calls the *CoreCompile* target (which *CCS.targets* defines) and then launches the debugger, this action is the same for all languages, although the compile step itself may be different.

The only additional thing done to make the debugging experience more user friendly was to implement a method named ValidateBreakpointLocation in the CCSLanguage class. When the user tries to set a breakpoint on a particular line, this method is called and is responsible for looking at the line and determining whether a breakpoint can be set there, and if so, which part of the line should be highlighted. For CCS the valid locations are actions, method calls, process invocations and expressions in `if` statements. Using the generated Scanner class once again, it is possible to find out which tokens are on a particular line and return their locations if they are valid points for a breakpoint. Figure 8.7 shows how a breakpoint is highlighted. (Note that this is only relevant for the highlighting done at compile time, at run time each line is highlighted according to the sequence points in the actual compiled file.)



```
use System.Console
use PLR.Runtime.BuiltIns

University = CoffeeMachine | ComputerScientist
CoffeeMachine      = coin  . _coffee_ . CoffeeMachine
At CCSTest.ccs, line 5 character 22 _coin_ . Co
```

Figure 8.7: Setting breakpoints in Visual Studio

8.3 Summary

There is no doubt that the tool support for programming languages is a factor in whether those languages become popular, how productive programmers are when working with them and how enjoyable they are to work with. Programming languages used in industry have for a long time now had great tool support while academic languages often suffer from bad or incomplete tools. As this chapter demonstrates, this need not be the case. Once a programming language (or any tool that takes some text files as input) has been developed, taking the extra time to develop a language service for it can be very beneficial to the end users of that language, and is relatively easy to do. One approach might even be to develop the language service first, thus enabling the author to benefit from it himself while writing the compiler. Visual Studio might not be the perfect environment for all languages, but in any case it is a good idea for the language author to research what environments are out there and evaluate if one of them could be used to offer the end user of the language a better user experience.

Chapter 9

Final Considerations

The first and second chapters of the thesis presented the objectives and goals of the project. They also introduced the subject matter, process algebras, their purpose and structure and what they had in common. The .NET framework was introduced, its history, relation to other virtual machine environments and current status was discussed, as well as the benefits of using virtual machines in general.

The third chapter then documented the design and implementation of the Process Language Runtime itself. The PLR self-compiling syntax tree was explained in detail and reasons given for why that was an optimal design for further extensibility. An overview of the PLR runtime classes was given, as well as an insight into how debugging support is enabled in compiled process language applications. The chapter ended by explaining in detail the structure of a .NET assembly compiled from the PLR syntax tree.

Chapter 4 presented a number of static analyses that are part of the PLR, both classical data flow analyses and more specific analyses for process algebra. How these analyses could be re-used for more than one process algebra, even when they have different constructs, was also explained.

Chapters 5 and 6 documented two separate implementations of process algebras using the PLR, the languages were CCS and KLAIM. The syntax and semantics of the languages was shown, as well as an overview of the most important classes in their implementation. The chapter on KLAIM then explored how additional features that are not included in the PLR could be added to new languages, by using custom syntax tree nodes, PLR compilation events and runtime libraries. Both chapters finished with sizable example systems to give an idea of what a real application of these languages might be.

In chapter seven we looked at a graphical user interface tool that allows the user to monitor and interact with running process language applications. The challenges faced when integrating with the PLR were discussed as well as how the program was kept general enough to work with any process language.

The eighth chapter was about how the CCS language was integrated into a professional development environment, Visual Studio 2008. Some alternative development environments were presented and reasons given for why Visual Studio was chosen. A number of useful features were implemented in the Visual Studio integration, including CCS project support, syntax highlighting, real-time syntax checking, IntelliSense, brace matching and full integration with the debugger and build tool.

Finally, in this chapter related work is discussed, including some alternative implementations of CCS and KLAIM. We explore the potential future work that could be done with the PLR, and end with some concluding remarks.

9.1 Related work

There are quite a few other projects that have implemented process algebras, or languages inspired by process algebras, however most of these projects take a different approach than that taken by the PLR. Below is a short summary of some of the notable ones, especially those that focus on CCS or KLAIM.

KLAVA [4] is an implementation of KLAIM. It is a Java library which represents the KLAIM constructs as Java classes. KLAIM applications can then be written in Java using the KLAVA library. The main difference between KLAVA and the PLR implementation of KLAIM (hereafter referred to as PLR KLAIM) is that in KLAVA it is not possible to write the applications using the actual syntax of KLAIM. KLAVA is a much more feature rich implementation of KLAIM than PLR KLAIM, it supports nodes running on different machines, and additional constructs such as non blocking input operations.

X-Klaim [3] (which stands for eXtended KLAIM) is another implementation of

KLAIM from the authors of KLAVA. It is at a higher level of abstraction than KLAVA and has its own syntax, which is a superset of the original KLAIM syntax. X-Klaim code uses the KLAVA library as its runtime library, the X-Klaim compiler compiles X-Klaim code down to Java code that uses the KLAVA library. X-Klaim is similar to PLR KLAIM in that both have a KLAIM syntax and both use a runtime library, the difference is that PLR KLAIM directly emits bytecodes, whereas X-Klaim emits Java source code, which means that X-Klaim code cannot be debugged using the original X-Klaim source files. X-Klaim is feature rich and supports many constructs not in the original KLAIM.

AspectK is an aspect oriented version of KLAIM. Originally introduced in [14], a full virtual machine for the language was subsequently developed in [2]. The KLAIM subset used in AspectK is the same as that used in PLR KLAIM, AspectK then adds aspects on top of that. The difference (aside from the aspect orientation) is that AspectK has its own virtual machine, with bytecodes for common process algebra tasks whereas PLR KLAIM uses an existing virtual machine. An advantage of having a process algebra focused virtual machine is that generated code can be smaller, since each bytecode instruction can perform more work. The PLR does get a similar reduction in code size by generating bytecodes that call methods defined in the PLR runtime library.

JACK [10] is a process algebra implementation written in Java. It is similar to the PLR in that it aims to be a framework that can be used for implementing different types of process algebra, although its main focus is Communicating Sequential Processes (CSP). The difference is that JACK, like KLAVA, represents algebra constructs as Java classes, and the systems are written using Java code instead of the native syntax of the process algebra being implemented. That is to say, it is a framework, but not a compiler.

CCS has at least two implementations, in [9] a method is presented for how to build a sound CCS interpreter by following the semantics of the language, and [15] shows how the functional programming language Haskell can be used to build a CCS interpreter with minimal amount of code. Both of these differ from the PLR in that they are interpreters rather than compilers.

None of the above related work aims to do exactly what the PLR attempts, which is to build compilers for process algebras that operate on the algebra's standard syntax and integrate tightly with an existing virtual machine. The PLR is also the only one of these that explores how existing infrastructure can be used to add features to process algebras, such as allowing CCS to call .NET methods written in another .NET language. A further look at that topic might prove interesting, specifically how it affects the original semantics of the algebra being implemented, what side effects it might produce and what sort of interesting things could be modeled in this way.

9.2 Further work

The PLR could be improved and built upon in several ways. Here we will look at some of them.

9.2.1 Additional process algebra constructs

Perhaps the most obvious improvement to the PLR would be to add support for some of the constructs that are common in process algebras but are not currently included in the PLR. This would make it even simpler to use the PLR as the basis for implementing other process algebras. To get a sense of which constructs would be most beneficial to add, we look shortly at two of the most prominent process algebras, CSP and π-calculus, and what would be needed for them to run on the PLR.

π-calculus was introduced in [24] and is described by its author as an extension of CCS. There are many variants of π-calculus with additional features but the core calculus has two noteworthy additions to CCS:

1. The *match* construct $[a = b]P$ which compares the values of a and b and behaves as P if they are equal but otherwise turns into the nil process. This is simply a more restricted version of the `if-then-else` construct which is already implemented in the PLR, the boolean condition is restricted to equality comparison and the `else` branch is simply the nil process.

2. The generalization of channel and variable names. In π-calculus both variable and channel names are seen simply as *names*, and can be passed along channels as data. For example a process P could send the channel name y to process Q, which could then send or receive on that channel.

$$\mathrm{P} \overset{\text{def}}{=} \overline{x}y.y(j).0$$
$$\mathrm{Q} \overset{\text{def}}{=} x(z).\overline{z}3.0$$

   Here we see that P first outputs the name y on channel x. Process Q receives the channel name on channel x, after which it substitutes z with y and becomes $\overline{y}3 \,.\, 0$. It then sends the value 3 on the received channel and P accepts it and binds it to the name j.

This could fairly easily be added to the PLR. At compilation time each channel name being used could be checked to see whether it was defined as a variable or not. If it had previously been defined as a variable then the synchronization would happen on the channel whose name was stored in the variable, if no variable with that name exists then the name of the channel would be considered a constant. For example in the term $x(z).\overline{z}3$ we see that z is bound in the first action, and so when we process the $\overline{z}3$ action we know that we should use the name stored in z as opposed to the literal name z. However in the term $x(z).\overline{y}3$ we see that y has never been bound as a variable and so when the $\overline{y}3$ action is processed the literal name y is used for the channel.

CSP is very similar to CCS. It can synchronize on channels, with or without message passing, it uses action prefixing, restriction, choice and parallel composition. Its definitions of choice and parallel composition are a little bit more complex than those of CCS. It also distinguishes between an inactive process, which is called $STOP$ and is equivalent to the nil process in CCS, and a $SKIP$ process which signals that a process has completed successfully. Finally, as its name *Communicating Sequential Processes* suggests, it offers sequential composition of processes. We now look at how these differences might be implemented in the PLR.

1. Implementing the $SKIP$ process is trivial since it does nothing. The main issue is distinguishing it from the $STOP$ process, this can be done by letting them output different text when they are invoked.

2. In CSP a distinction is made between internal non deterministic choice (written $P \sqcap Q$) and external non deterministic choice (written $P \,\Box\, Q$). External choice will synchronize with the first event offered by the environment, so in $a.STOP \,\Box\, b.STOP$ it depends on which of a and b is offered first. Internal non deterministic choice however can refuse to participate in an event even though no alternative is offered. The current implementation in the PLR is equivalent to CSP's external choice, if an event (or channel synchronization) is offered on one of the paths then it will be taken. If two or more candidates are offered then the selection between them is made randomly. To enable a simulation of internal choice it could be possible to add some randomness to whether or not an offered channel synchronization is accepted. When the scheduler is finding out which synchronizations are possible it calls a CanSyncWith method on each candidate action. It could be possible to make that call randomly return true or false, which would make sure that a path in an internal choice was not forced to be taken, even if it was the only candidate for synchronization.

3. CSP also handles parallel composition in a slightly different way from CSS. It defines two versions of parallel composition. The first one, which is sometimes called *interleaving* is written as $P \mid\mid\mid Q$. Here P and Q run in parallel and are independent of each other. They can interact but are not forced to. This is equivalent to the PLR's parallel composition construct.

   The other type of parallel composition available in CSP is written as $P \mid\mid_A Q$ or sometimes as $P \mid[\{A\}]\mid Q$. P and Q are said to be *interface parallel*. Here A is a set of channel names that P and Q must synchronize on, e.g. in $P \mid[\{a, b\}]\mid Q$ the processes P and Q must synchronize with each others on channels a and b. If one of them has arrived at an a or b action it cannot continue until the other one is ready to synchronize with it. This is similar to parallel composition with restriction in CCS, e.g. $(P \mid Q)\backslash\{a, b\}$. In that expression P and Q have to synchronize on a and b because they are invisible from the outside and so for either process the other process is the only candidate to synchronize with. It is slightly different though because in CSP's version the events a and b are not unobservable from the outside. This could still be implemented much like the CCS expression shown above, the channels a and b would be locally scoped to the $P \mid[\{a, b\}]\mid Q$ process so that no external processes could participate in the synchronization, and the channels would be shown as part of the trace of the system, which would not normally be done in an expression like $(P \mid Q)\backslash\{a, b\}$

4. Sequential composition is written as $P; Q$, it is a process that behaves like P until P terminates and then behaves like Q. This would be easy to implement in the PLR, any finite process will end up as the nil process (or the $STOP$ or $SKIP$ process in CSP), once the nil process has been reached in P a new instance of Q could be instantiated and started.

In addition to these features, variants of CSP sometimes contain additional features such as interrupts and timeouts which we shall not go into here.

9.2.2 Richer datatypes and expressions

The initial version of the PLR supports two types of variables, integers and strings. For expressions it supports constants for integers, booleans and strings, as well as simple arithmetic and relational operators for integers and logical operators for booleans. One way to extend the PLR would be to add more support for using other data types from .NET. An example would be to allow passing of .NET objects through channels and calling instance methods on those objects in the receiving process. This would require some additional syntax nodes for the PLR tree, an expression node for constructing a new object and a node for a method call on an object (as opposed to a static method call which is already supported). Other useful features to add might include support for floating point numbers, string formatting and basic string expressions using the + operator.

9.2.3 Additional analysis

Another potential improvement would be to add additional analyses before compilation. This is where the benefit of having a shared syntax tree for multiple algebras becomes apparent, as many of the analyses could be re-used for multiple process algebras (although probably not all of them). This could include common compiler optimization techniques such as constant propagation and Very Busy Expressions analysis, or analyses more directly related to process algebra, such as finding channels that are never used and identifying processes that will always block.

9.2.4 Bi-similarity of processes

One of the interesting things that could be added, for CCS and maybe others, would be an analysis to compare two processes and see if they are *behaviorally equivalent*, also known as *bi-similar*. B-similarity is a congruence, if processes P and Q are bi-similar then it means that if P is a component in a system then it can be replaced with Q and the system will continue to work in the same way, since P and Q exhibit the same behavior. This can for instance be used to write a specification as a simple process expression and then write an implementation for that specification. If the specification and implementation are bi-similar then the implementation is a correct implementation of the specification. An example could be the specification

$$\text{CoffeeMachine} \stackrel{\text{def}}{=} coin\,.\,\overline{coffee}\,.\,CoffeeMachine$$

and the implementation

$$\begin{aligned}\text{CoffeeMachineImpl} &\stackrel{\text{def}}{=} (\text{CoinReceiver} \mid \text{CoffeePourer})\backslash\{pour\}\\ \text{CoinReceiver} &\stackrel{\text{def}}{=} coin\,.\,\overline{pour}\,.\,\text{CoinReceiver}\\ \text{CoffeePourer} &\stackrel{\text{def}}{=} pour\,.\,\overline{coffee}\,.\,\text{CoffeePourer}\end{aligned}$$

Here we see that according to the specification of CoffeeMachine the observable events are an endless stream of *coin* and $coffee$. This however tells us nothing about how this machine is implemented. The second process CoffeeMachineImpl is the implementation of this coffee machine, it is composed of two components, a receiver for the coins and a component that pours the coffee. They communicate between themselves on the *pour* channel. Since that channel is hidden (or restricted) it is not observable from the outside, what is observable from the outside is again an endless stream of *coin* and $coffeee$. In this trivial case it is obvious that CoffeeMachineImpl is a valid implementation of CoffeeMachine.

Bi-similarity can be analyzed by converting a CCS process expression into a *labelled transition system*, which is a state machine where the transitions between states are the actions performed in the process. Weak bi-simulation, or observational equivalence, is perhaps the most interesting bi-simulation to verify. In general terms it states that if P and Q are weakly bi-similar then they will behave exactly the same when observed from the outside, they will offer the same synchronizations or events. However before and after these public events they can perform any number of internal actions or τ actions which do not have to match between the two processes since they do not affect their behavior as seen from the outside.

The PLR syntax tree is a rich data structure and would be well suited for this type of analysis. This might not be the type of feature that belongs in a process algebra compiler, instead it might be incorporated into some kind of analysis tool for CCS (or other process algebras) that could make use of the syntax tree of the compiler and the parser and scanner of the CCS compiler.

This section has only briefly touched on the possibilities of verifying process behavior using bi-simulation, for a more comprehensive explanation see [1].

9.3 Conclusions

The initial goal of this project was to explore how process algebras could fit in with the .NET framework and how the constructs these algebras had in common could be abstracted into common building blocks that could be re-used in many implementations. The implementations of two process algebras using those building blocks was necessary to prove their generality. Integrating a process algebra into Visual Studio was more of an afterthought, but once I started on it I saw how useful it was, and how much more enjoyable it was to write CCS in this integrated environment, and so I was inspired to explore exactly how well the language could be integrated and what services were available.

After finishing this project I believe that the .NET framework is a good platform for implementing process algebras. The main benefit is the ease of interacting with other .NET languages. It is easy to add additional features to a language by writing runtime libraries in languages like C# and it is easy to make a process algebra call into any arbitrary .NET assembly. Another great benefit is being able to debug the compiled applications, and in general great tool support in programs like Visual Studio. The built in API to emit bytecode is well structured and easy to use, and the bytecode itself is well designed and surprisingly readable once you have gotten used to it.

There are downsides to .NET as well. The only real support for concurrency is by using threads. I had thought beforehand that there might be some low level support for that in the actual bytecode but that was not the case. Threading can only be done through the standard class library, using the Thread class, which shows that the class library is at least as important as the virtual machine itself. It is also my conclusion that while .NET is a good platform for implementing process algebras, it is not *specifically* good for process algebras, it is more that it is a good platform for implementing programming languages in general.

The fact that there exist other implementations of the CLI (Common Language Infrastructure) specification is very useful. The PLR library, CCS compiler and KLAIM compiler all work flawlessly on Mono, the CLI implementation that runs on the Linux and Unix family of operating systems. The compiled process applications also work on Mono without problems. I had expected that the compilers and the PLR would work, since they are command line tools/libraries and do not depend on the underlying operating system much, except for reading and writing files. What I did not expect was that the Process Viewer application would work, since it has a graphical user interface which uses the underlying graphic system of the Windows operating system. To my surprise the Process Viewer ran without any problems on Mono the first time I tried it. It is worth noting that no special consideration was needed to support Mono, I simply wrote the whole thing using .NET on Windows and once it was ready I compiled and ran it with Mono 2.4.2 on an Ubuntu Linux 8.10 operating system and it worked flawlessly. This means that the PLR is truly cross-platform, which should make it accessible to more people, especially since many researchers in computer science do not use the Windows operating system at all.

Building re-usable components for process algebras went very well. The PLR takes care of a lot of the basic work that anyone implementing a process algebra would otherwise have to do themselves. Not just the basic process algebra constructs themselves, but also expression trees for arithmetic expressions, static analyses, emitting debugging symbols, and helper methods for calling into .NET code. An implementor can use this and focus his energy on what matters, implementing specific constructs and emitting the bytecodes for them.

Overall I consider the project a success, as it has resulted in two working process algebra implementations (one of which is fully integrated into Visual Studio), a graphical tool for working with compiled process language applications, as well as the main software product: a library/compiler that can (and hopefully will) be used by future implementers of other process algebras.

Appendix A

Software

During the course of this project four distinct software packages were developed, a CCS compiler, a CCS integration package for Visual Studio, a KLAIM compiler and the Process Viewer application. Here we look at the practical aspects of this software, where it can be downloaded, how it is licensed and how it can be built, configured and used.

A.1 Licensing and availability

All the software developed during this project can be downloaded from the url `http://einaregilsson.com/plr`. The source code for the entire project is available in a zip file. The binaries for each of the software packages can be downloaded seperately.

The source code for the project is licensed under the General Public License (GPL) v3.0. In brief, this allows anyone to download and modify the source or use it as a basis for something else, as long as the source code for that modified version is also made available under a GPL compatible license. For further information see `http://www.gnu.org/licenses/gpl.html`.

A.2 Source code

The source code is organized into two solutions and ten projects. They are as follows:

- **MSC** is the solution for the compilers and runtimes.
  - **PLR** is the project for the Process Language Runtime.
  - **CCS** is the project for the CCS compiler.
  - **CCS.External** is a project which contains utility functions written in C# which can be called from CCS if the CCS.External.dll is referenced during compilation.
  - **KLAIM** is the project for the KLAIM compiler.
  - **KLAIM.Runtime** is the project for the KLAIM runtime library.
  - **ProcessViewer** is the project for the Process Viewer visualization tool.
- **CCS.Integration** is the solution for Visual Studio Integration.
  - **CCS.BuildTasks** contains MSBuild build tasks for the CCS compiler.
  - **CCS.LanguageService** is the main integration project, it contains the Visual Studio language service for CCS.
  - **CCS.Projects** contains the Visual Studio package that allows CCS projects to be created in Visual Studio.
  - **CCS.Deployment** is a project that builds an installer for the entire integration package.

The solutions and projects can be built using Visual Studio 2008, although that is not required. The C# compiler and MSBuild build tool are included with the standard .NET framework distribution, they are enough to build the projects. To build the entire MSC solution, first add the location where MSBuild is stored to the PATH environment variable, this can be done with

```
SET PATH=%PATH%;c:\Windows\Microsoft.NET\Framework\v3.5
```

The solution can then be built with the command

```
msbuild MSC.sln /p:Configuration=Debug /p:Platform="Any CPU"
```

The .NET framework v3.5 is required to build both solutions. To build the CCS.Integration solution it is necessary to have the Visual Studio 2008 SDK installed.

A.3 CCS Compiler

The CCS compiler is an executable file named `ccs.exe`. It has one dependency which is the PLR itself, it is in a file named `PLR.dll`. The compiler is a command line tool and is invoked as

```
ccs.exe [options] <filename>
```

It accepts one input file (`<filename>`) and can accept a number of optional command line switches (`[options]`). By default the generated executable file will have the same name as the input file, except ending with `.exe` instead of `.ccs`. The compiled file will have a dependency on the PLR for the runtime system. To get a guide to the available command line options the compiler can be invoked as `ccs.exe /?` . The output of that command is shown below:

```
CCS Compiler
Copyright (C) 2009 Einar Egilsson

Usage: CCS [options] <filename>

Available options:

    /reference:<files>   The assemblies that this program requires. It is
    /r:<files>           not neccessary to specify the PLR assembly.
                         Other assemblies should be specified in a comma
                         seperated list, e.g. /reference:Foo.dll,Bar.dll.

    /optimize            If specified then the generated assembly will be
    /op                  optimized, dead code eliminated and expressions
                         pre-evaluated where possible. Do not combine this
                         with the /debug switch.

    /embedPLR            Embeds the PLR into the generated file, so it can
    /e                   be distributed as a stand-alone file.

    /debug               Emit debugging symbols in the generated file,
    /d                   this allows it to be debugged in Visual Studio, or
                         in the free graphical debugger that comes with the
                         .NET Framework SDK.

    /out:<filename>      Specify the name of the compiled executable. If
    /o:<filename>        this is not specified then the name of the input
```

```
                       file is used, with .ccs replaced by .exe.

/print:<format>        Prints a version of the program source in the
/p:<format>            specified format. Allowed formats are ccs, html
                       and latex. The generated file will have the same
                       name as the input file, except with the format
                       as extension.
```

A.4 CCS Visual Studio Integration Package

The integration package for CCS can be downloaded as a MSI installer. To start using the integration package simply follow the instructions in the installer program, it will install the necessary files and register the language service with Visual Studio.

After installing the integration package the following features will be added to Visual Studio:

- When creating a new project there is an option named *CCS Project*. Choosing this option creates a new project with the file ending .ccsproj and includes references to the PLR and a CCS source code file with some example code.
- The CCS project can be built using Visual Studio's *Build* command or by using the keyboard shortcut Ctrl+Shift+B.
- Any file that is edited in Visual Studio which has the file ending .ccs will be handled by the language service, which will syntax highlight it and warn about syntax errors.
- IntelliSense can be invoked in .ccs files by pressing Ctrl+Space.
- If the *Debug* configuration is chosen then a CCS system can be debugged by pressing the "play" button, or by pressing F5.

The integration package can be uninstalled in the standard Windows *Add/Remove programs* dialog.

A.5 KLAIM Compiler

The KLAIM compiler is an executable file named `kc.exe`. It has two dependencies, the KLAIM runtime (`KlaimRuntime.dll`) and the PLR itself (`PLR.dll`). The KLAIM compiler is very similar to the CCS compiler, it is a command line tool that is invoked as

```
kc.exe [options] <filename>
```

It accepts one input file (`<filename>`) and can accept a number of optional command line switches (`[options]`). The generated executable file will have the same name as the input file, except ending with `.exe` instead of `.klaim`. The command `kc.exe /?` will print out the available options and then exit. The output of that command is shown below:

```
KLAIM Compiler
Copyright (C) 2009 Einar Egilsson

Usage: kc [options] <filename>

Available options:

    /optimize           If specified then the generated assembly will be
    /op                 optimized, dead code eliminated and expressions
                        pre-evaluated where possible. Do not combine this
                        with the /debug switch.

    /debug              Emit debugging symbols in the generated file,
    /d                  this allows it to be debugged in Visual Studio, or
                        in the free graphical debugger that comes with the
                        .NET Framework SDK.

    /embedKLAIM         Embeds the KLAIM runtime into the generated file,
    /ek                 so it does not need to be distributed with the
                        executable.

    /embedPLR           Embeds the PLR into the generated file, so it does
    /e                  not need to be distributed with the executable.

    /out:<filename>     Specify the name of the compiled executable. If
    /o:<filename>       this is not specified then the name of the input
                        file is used, with .ccs replaced by .exe.
```

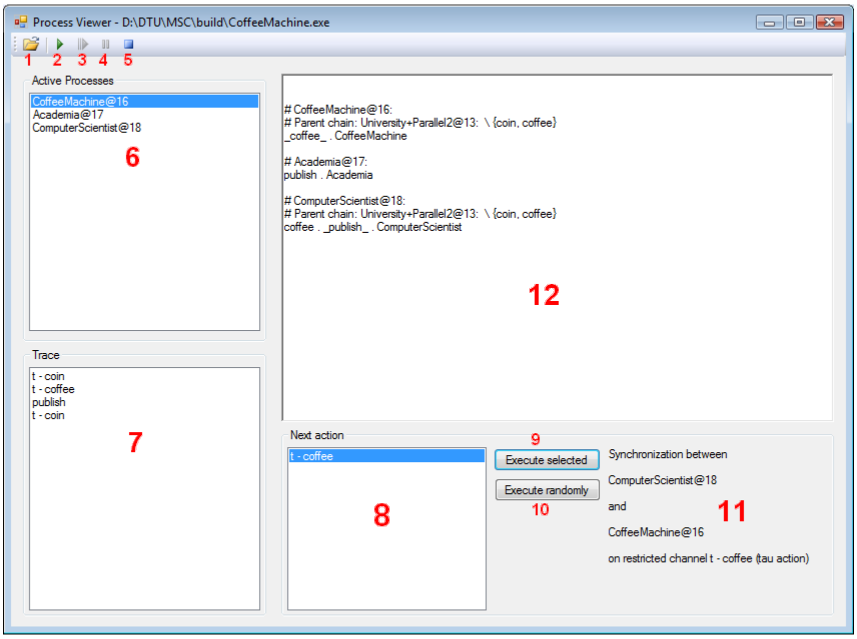
A.6 Process Viewer

The Process Viewer is an executable named `ProcessViewer.exe`. It requires the PLR library, `PLR.dll` as well as any assemblies that contain parsers for the languages being used. The filenames of the parser assemblies are specified in the configuration file, `ProcessViewer.exe.config`. An example is

```
<add key="ParserAssemblies" value="CCS.exe;kc.exe"/>
```

Here there are two assemblies specified, their names seperated by a semicolon.

Figure A.1 shows the main screen, with the main controls labelled with numbers. Below each number corresponding to a control is explained.



CoffeeMachine@16
Academia@17
ComputerScientist@18

```
# CoffeeMachine@16:
# Parent chain: University+Parallel2@13: \ {coin, coffee}
_coffee_ . CoffeeMachine

# Academia@17:
publish . Academia

# ComputerScientist@18:
# Parent chain: University+Parallel2@13: \ {coin, coffee}
coffee . _publish_ . ComputerScientist
```

t - coin
t - coffee
publish
t - coin

t - coffee

Synchronization between
ComputerScientist@18
and
CoffeeMachine@16
on restricted channel t - coffee (tau action)

Figure A.1: The Process Viewer application running

1. The *open* button. Press this to open a new process language executable file, and optionally its original source file.
2. Starts execution of the loaded process language application.
3. Starts execution of the loaded process language application in step mode. This means that the user will select each action that is executed.

4. Pauses execution. This will put a running application into step mode.
5. Stops execution of running application.
6. This is a list of all active processes at the current time.
7. The trace is the list of actions that have been executed.
8. This is a list of the actions that could potentially be executed next.
9. Pressing this button executes the currently selected action.
10. Pressing this button executes a random action.
11. Additional information about the selected action is displayed here.
12. The current state of the active processes is shown here, along with their parent chains, that is the restricted processes that spawned them and are kept alive so that the restrictions still apply.

Appendix B

Generated bytecode

For those that are interested in the actual code generation that takes place in the PLR, we now look at a sample system and its generated code. The system we look at is the following:

```
use PLR.Runtime.BuiltIns

StartProc = ActionPrefix | NonDeterministicChoice

ActionPrefix = a .  0

ValuePassSend(x) = _a_(x+3 /:Rand(2)) . 0
ValuePassReceive = a(y) . 0

NonDeterministicChoice = _a_ . 0 + NDC2
NDC2 = b . 0

ParallelComposition = a . 0 | PC2 | 0
PC2 = b . 0

MethodCall = :Print("Hello") . 0
```

```
Restrict = ( a . (d . 0)\ d ) \{a}

Relabel = ( a . (d . 0)[dnew/d] )[anew/a]
```

This system obviously is not a model of anything special, it is simply composed to show the available features of the PLR, and the processes are named accordingly, e.g. ActionPrefix and Restrict. In this appendix each activity type will have its own section, where the relevant process is first shown in CCS and then the generated bytecode is shown. The text representation of the bytecode was generated using the CIL disassembler tool from Microsoft, *ILDASM*.

B.1 Header

The header of the CIL file is as follows:

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 3.5.21022.8
//  Copyright (c) Microsoft Corporation.  All rights reserved.

// Metadata version: v2.0.50727
.assembly extern PLR
{
  .ver 1:0:0:0
}
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )        // .z\V.4..
  .ver 2:0:0:0
}
.assembly disassemble.exe
{
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module disassemble.exe
// MVID: {A23EA4EB-468E-4E2B-A4CA-AA68D8C2320E}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x00970000
```

B.2 Main method

The main method of the executable, which is the entry point, is as follows:

```
// ================== GLOBAL METHODS =========================

.method public static int32  Main() cil managed
{
  .entrypoint
  // Code size       18 (0x12)
  .maxstack  1
  .locals init (class StartProc V_0)
  IL_0000:  newobj     instance void StartProc::.ctor()
  IL_0005:  stloc.0
  IL_0006:  call       class [PLR]PLR.Runtime.Scheduler
                       [PLR]PLR.Runtime.Scheduler::get_Instance()
  IL_000b:  call       instance void [PLR]PLR.Runtime.Scheduler::Run()
  IL_0010:  ldc.i4.0
  IL_0011:  ret
} // end of global method Main


// =============================================================
```

B.3 StartProc

```
StartProc = ActionPrefix | NonDeterministicChoice
```

is compiled as follows:

```

.class public auto ansi beforefieldinit StartProc
       extends [PLR]PLR.Runtime.ProcessBase
{
  .method public specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase::.ctor()
    IL_0006:  ret
  } // end of method StartProc::.ctor

  .method public virtual instance void  RunProcess() cil managed
  {
    .override [PLR]PLR.Runtime.ProcessBase::RunProcess
    // Code size       63 (0x3f)
    .maxstack  2
    .locals init ([0] class [PLR]PLR.Runtime.ProcessBase V_0,
             [1] class [PLR]PLR.Runtime.ProcessBase V_1)
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::InitSetID()
    IL_0006:  nop
    IL_0007:  newobj     instance void ActionPrefix::.ctor()
    IL_000c:  stloc.0
    IL_000d:  ldloc.0
    IL_000e:  ldarg.0
    IL_000f:  call       instance class [PLR]PLR.Runtime.ProcessBase
```

```
                        [PLR]PLR.Runtime.ProcessBase::get_Parent()
    IL_0014:  call       instance void [PLR]PLR.Runtime.ProcessBase
                        ::set_Parent(class [PLR]PLR.Runtime.ProcessBase)
    IL_0019:  ldloc.0
    IL_001a:  call       instance void [PLR]PLR.Runtime.ProcessBase::Run()
    IL_001f:  nop
    IL_0020:  newobj     instance void NonDeterministicChoice::.ctor()
    IL_0025:  stloc.1
    IL_0026:  ldloc.1
    IL_0027:  ldarg.0
    IL_0028:  call       instance class [PLR]PLR.Runtime.ProcessBase
                        [PLR]PLR.Runtime.ProcessBase::get_Parent()
    IL_002d:  call       instance void [PLR]PLR.Runtime.ProcessBase
                        ::set_Parent(class [PLR]PLR.Runtime.ProcessBase)
    IL_0032:  ldloc.1
    IL_0033:  call       instance void [PLR]PLR.Runtime.ProcessBase::Run()
    IL_0038:  ldarg.0
    IL_0039:  call       instance void [PLR]PLR.Runtime.ProcessBase::Die()
    IL_003e:  ret
  } // end of method StartProc::RunProcess

} // end of class StartProc
```

B.4 ActionPrefix

```
  ActionPrefix = a .  0
```

is compiled as follows:

```
.class public auto ansi beforefieldinit ActionPrefix
       extends [PLR]PLR.Runtime.ProcessBase
{
  .method public specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase::.ctor()
    IL_0006:  ret
  } // end of method ActionPrefix::.ctor

  .method public virtual instance void  RunProcess() cil managed
  {
    .override [PLR]PLR.Runtime.ProcessBase::RunProcess
    // Code size       84 (0x54)
    .maxstack  10
    .locals init ([0] class [PLR]PLR.Runtime.ChannelSyncAction V_0)
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                        ::InitSetID()
    .try
    {
      IL_0006:  ldarg.0
      IL_0007:  ldstr      "Preparing to sync now..."
      IL_000c:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0011:  ldarg.0
      IL_0012:  ldstr      "a"
      IL_0017:  ldarg.0
      IL_0018:  ldc.i4     0x0
      IL_001d:  ldc.i4.1
      IL_001e:  newobj     instance void [PLR]PLR.Runtime.ChannelSyncAction
                           ::.ctor(string, class [PLR]PLR.Runtime.ProcessBase
                                   ,int32, bool)
      IL_0023:  stloc.0
      IL_0024:  ldloc.0
      IL_0025:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Sync(class [PLR]PLR.Runtime.IAction)
      IL_002a:  nop
      IL_002b:  nop
      IL_002c:  ldarg.0
      IL_002d:  ldstr      "Turned into 0"
      IL_0032:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0037:  leave      IL_004d

    }  // end .try
    catch [PLR]PLR.Runtime.ProcessKilledException
    {
      IL_003c:  pop
      IL_003d:  ldarg.0
      IL_003e:  ldstr      "Caught ProcessKilledException"
      IL_0043:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0048:  leave      IL_004d

    }  // end handler
    IL_004d:  ldarg.0
    IL_004e:  call       instance void [PLR]PLR.Runtime.ProcessBase::Die()
    IL_0053:  ret
  } // end of method ActionPrefix::RunProcess

} // end of class ActionPrefix
```

B.5 ValuePassSend and ValuePassReceive

```
ValuePassSend(x) = _a_(x+3 /:Rand(2)) . 0
ValuePassReceive = a(y) . 0
```

are compiled as follows:

```
.class public auto ansi beforefieldinit ValuePassSend_1
       extends [PLR]PLR.Runtime.ProcessBase
{
  .field assembly object x
  .method public specialname rtspecialname
          instance void  .ctor(object x) cil managed
  {
    // Code size       19 (0x13)
    .maxstack  4
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase::.ctor()
    IL_0006:  ldarg.0
    IL_0007:  ldarg      x
    IL_000b:  nop
    IL_000c:  nop
    IL_000d:  stfld      object ValuePassSend_1::x
    IL_0012:  ret
  } // end of method ValuePassSend_1::.ctor

  .method public virtual instance void  RunProcess() cil managed
  {
    .override [PLR]PLR.Runtime.ProcessBase::RunProcess
    // Code size       125 (0x7d)
    .maxstack  10
    .locals init ([0] object x,
             [1] class [PLR]PLR.Runtime.ChannelSyncAction V_1)
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::InitSetID()
    .try
    {
      IL_0006:  ldarg.0
      IL_0007:  ldfld      object ValuePassSend_1::x
      IL_000c:  stloc.0
      IL_000d:  ldarg.0
      IL_000e:  ldstr      "Preparing to sync now..."
      IL_0013:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0018:  ldarg.0
      IL_0019:  ldstr      "a"
      IL_001e:  ldarg.0
      IL_001f:  ldc.i4     0x1
      IL_0024:  ldc.i4.0
      IL_0025:  newobj     instance void [PLR]PLR.Runtime.ChannelSyncAction
                           ::.ctor(string,class [PLR]PLR.Runtime.ProcessBase,
                           int32, bool)
      IL_002a:  stloc.1
      IL_002b:  ldloc.1
      IL_002c:  ldloc.0
      IL_002d:  unbox.any  [mscorlib]System.Int32
      IL_0032:  ldc.i4     0x3
      IL_0037:  ldc.i4     0x2
      IL_003c:  call       int32 [PLR]PLR.Runtime.BuiltIns::Rand(int32)
      IL_0041:  div
      IL_0042:  add.ovf
      IL_0043:  box        [mscorlib]System.Int32
      IL_0048:  call       instance void [PLR]PLR.Runtime.ChannelSyncAction
                           ::AddValue(object)
      IL_004d:  ldloc.1
      IL_004e:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Sync(class [PLR]PLR.Runtime.IAction)
      IL_0053:  nop
      IL_0054:  nop
      IL_0055:  ldarg.0
      IL_0056:  ldstr      "Turned into 0"
      IL_005b:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0060:  leave      IL_0076

    }  // end .try
    catch [PLR]PLR.Runtime.ProcessKilledException
    {
      IL_0065:  pop
      IL_0066:  ldarg.0
      IL_0067:  ldstr      "Caught ProcessKilledException"
      IL_006c:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0071:  leave      IL_0076

    }  // end handler
    IL_0076:  ldarg.0
    IL_0077:  call       instance void [PLR]PLR.Runtime.ProcessBase::Die()
    IL_007c:  ret
  } // end of method ValuePassSend_1::RunProcess

} // end of class ValuePassSend_1

.class public auto ansi beforefieldinit ValuePassReceive
       extends [PLR]PLR.Runtime.ProcessBase
{
  .method public specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase::.ctor()
    IL_0006:  ret
  } // end of method ValuePassReceive::.ctor

  .method public virtual instance void  RunProcess() cil managed
  {
    .override [PLR]PLR.Runtime.ProcessBase::RunProcess
    // Code size       96 (0x60)
    .maxstack  10
    .locals init ([0] class [PLR]PLR.Runtime.ChannelSyncAction V_0,
             [1] object y)
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::InitSetID()
    .try
    {
      IL_0006:  ldarg.0
      IL_0007:  ldstr      "Preparing to sync now..."
      IL_000c:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0011:  ldarg.0
      IL_0012:  ldstr      "a"
      IL_0017:  ldarg.0
      IL_0018:  ldc.i4     0x1
      IL_001d:  ldc.i4.1
      IL_001e:  newobj     instance void [PLR]PLR.Runtime.ChannelSyncAction
                           ::.ctor(string,class [PLR]PLR.Runtime.ProcessBase,
                           int32, bool)
      IL_0023:  stloc.0
      IL_0024:  ldloc.0
      IL_0025:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Sync(class [PLR]PLR.Runtime.IAction)
      IL_002a:  nop
      IL_002b:  ldloc.0
      IL_002c:  ldc.i4     0x0
      IL_0031:  call       instance object [PLR]PLR.Runtime.ChannelSyncAction
                           ::GetValue(int32)
      IL_0036:  stloc.1
      IL_0037:  nop
      IL_0038:  ldarg.0
      IL_0039:  ldstr      "Turned into 0"
      IL_003e:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0043:  leave      IL_0059

    }  // end .try
    catch [PLR]PLR.Runtime.ProcessKilledException
    {
      IL_0048:  pop
      IL_0049:  ldarg.0
      IL_004a:  ldstr      "Caught ProcessKilledException"
      IL_004f:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0054:  leave      IL_0059

    }  // end handler
    IL_0059:  ldarg.0
    IL_005a:  call       instance void [PLR]PLR.Runtime.ProcessBase::Die()
    IL_005f:  ret
  } // end of method ValuePassReceive::RunProcess

} // end of class ValuePassReceive
```

B.6 NonDeterministicChoice

```
NonDeterministicChoice = _a_ . 0 + NDC2
NDC2 = b . 0
```

are compiled as follows:

```
.class public auto ansi beforefieldinit NonDeterministicChoice
       extends [PLR]PLR.Runtime.ProcessBase
{
  .class auto ansi nested public beforefieldinit NonDeterministic1
         extends [PLR]PLR.Runtime.ProcessBase
  {
    .method public virtual instance void
            RunProcess() cil managed
    {
      .override [PLR]PLR.Runtime.ProcessBase::RunProcess
      // Code size       84 (0x54)
      .maxstack  10
      .locals init ([0] class [PLR]PLR.Runtime.ChannelSyncAction V_0)
      IL_0000:  ldarg.0
      IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::InitSetID()
      .try
      {
        IL_0006:  ldarg.0
        IL_0007:  ldstr      "Preparing to sync now..."
        IL_000c:  call       instance void [PLR]PLR.Runtime.ProcessBase
                             ::Debug(string)
        IL_0011:  ldarg.0
        IL_0012:  ldstr      "a"
        IL_0017:  ldarg.0
        IL_0018:  ldc.i4     0x0
        IL_001d:  ldc.i4.0
        IL_001e:  newobj     instance void [PLR]PLR.Runtime.ChannelSyncAction
                             ::.ctor(string,
                              class [PLR]PLR.Runtime.ProcessBase, int32, bool)
        IL_0023:  stloc.0
        IL_0024:  ldloc.0
        IL_0025:  call        instance void [PLR]PLR.Runtime.ProcessBase
                              ::Sync(class [PLR]PLR.Runtime.IAction)
        IL_002a:  nop
        IL_002b:  nop
        IL_002c:  ldarg.0
        IL_002d:  ldstr       "Turned into 0"
        IL_0032:  call        instance void [PLR]PLR.Runtime.ProcessBase
                              ::Debug(string)
        IL_0037:  leave       IL_004d

      }  // end .try
      catch [PLR]PLR.Runtime.ProcessKilledException
      {
        IL_003c:  pop
        IL_003d:  ldarg.0
        IL_003e:  ldstr       "Caught ProcessKilledException"
        IL_0043:  call        instance void [PLR]PLR.Runtime.ProcessBase
                              ::Debug(string)
        IL_0048:  leave       IL_004d

      }  // end handler
      IL_004d:  ldarg.0
      IL_004e:  call        instance void [PLR]PLR.Runtime.ProcessBase::Die()
      IL_0053:  ret
    } // end of method NonDeterministic1::RunProcess

    .method public specialname rtspecialname
            instance void  .ctor() cil managed
    {
      // Code size       7 (0x7)
      .maxstack  2
      IL_0000:  ldarg.0
      IL_0001:  call        instance void [PLR]PLR.Runtime.ProcessBase
                            ::.ctor()
      IL_0006:  ret
    } // end of method NonDeterministic1::.ctor

  } // end of class NonDeterministic1

  .method public specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  call        instance void [PLR]PLR.Runtime.ProcessBase
                          ::.ctor()
    IL_0006:  ret
  } // end of method NonDeterministicChoice::.ctor

  .method public virtual instance void  RunProcess() cil managed
  {
    .override [PLR]PLR.Runtime.ProcessBase::RunProcess
    // Code size       85 (0x55)
    .maxstack  2
    .locals init (class [PLR]PLR.Runtime.ProcessBase V_0,
             class [PLR]PLR.Runtime.ProcessBase V_1)
    IL_0000:  ldarg.0
    IL_0001:  call        instance void [PLR]PLR.Runtime.ProcessBase
                          ::InitSetID()
    IL_0006:  newobj      instance void
                          NonDeterministicChoice/NonDeterministic1::.ctor()
    IL_000b:  stloc.0
    IL_000c:  ldloc.0
    IL_000d:  ldarg.0
    IL_000e:  call       instance class [PLR]PLR.Runtime.ProcessBase
                         [PLR]PLR.Runtime.ProcessBase::get_Parent()
    IL_0013:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::set_Parent(class [PLR]PLR.Runtime.ProcessBase)
    IL_0018:  ldloc.0
    IL_0019:  ldarg.0
    IL_001a:  call       instance valuetype [mscorlib]System.Guid
                         [PLR]PLR.Runtime.ProcessBase::get_SetID()
    IL_001f:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::set_SetID(valuetype [mscorlib]System.Guid)
    IL_0024:  ldloc.0
    IL_0025:  call       instance void [PLR]PLR.Runtime.ProcessBase::Run()
    IL_002a:  newobj     instance void NDC2::.ctor()
    IL_002f:  stloc.1
    IL_0030:  ldloc.1
    IL_0031:  ldarg.0
    IL_0032:  call       instance class [PLR]PLR.Runtime.ProcessBase
                         [PLR]PLR.Runtime.ProcessBase::get_Parent()
    IL_0037:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::set_Parent(class [PLR]PLR.Runtime.ProcessBase)
    IL_003c:  ldloc.1
    IL_003d:  ldarg.0
    IL_003e:  call       instance valuetype [mscorlib]System.Guid
                         [PLR]PLR.Runtime.ProcessBase::get_SetID()
    IL_0043:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::set_SetID(valuetype [mscorlib]System.Guid)
    IL_0048:  ldloc.1
    IL_0049:  call       instance void [PLR]PLR.Runtime.ProcessBase::Run()
    IL_004e:  ldarg.0
    IL_004f:  call       instance void [PLR]PLR.Runtime.ProcessBase::Die()
    IL_0054:  ret
  } // end of method NonDeterministicChoice::RunProcess

} // end of class NonDeterministicChoice

.class public auto ansi beforefieldinit NDC2
       extends [PLR]PLR.Runtime.ProcessBase
{
  .method public specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase::.ctor()
    IL_0006:  ret
  } // end of method NDC2::.ctor

  .method public virtual instance void  RunProcess() cil managed
  {
    .override [PLR]PLR.Runtime.ProcessBase::RunProcess
    // Code size       84 (0x54)
    .maxstack  10
    .locals init ([0] class [PLR]PLR.Runtime.ChannelSyncAction V_0)
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::InitSetID()
    .try
    {
      IL_0006:  ldarg.0
      IL_0007:  ldstr      "Preparing to sync now..."
      IL_000c:  call       instance void [PLR]PLR.Runtime.ProcessBase
```

```
                        ::Debug(string)
      IL_0011:  ldarg.0
      IL_0012:  ldstr       "b"
      IL_0017:  ldarg.0
      IL_0018:  ldc.i4      0x0
      IL_001d:  ldc.i4.1
      IL_001e:  newobj      instance void [PLR]PLR.Runtime.ChannelSyncAction
                            ::.ctor(string,class [PLR]PLR.Runtime.ProcessBase,
                            int32, bool)
      IL_0023:  stloc.0
      IL_0024:  ldloc.0
      IL_0025:  call        instance void [PLR]PLR.Runtime.ProcessBase
                            ::Sync(class [PLR]PLR.Runtime.IAction)
      IL_002a:  nop
      IL_002b:  nop
      IL_002c:  ldarg.0
      IL_002d:  ldstr       "Turned into 0"
      IL_0032:  call        instance void [PLR]PLR.Runtime.ProcessBase
                            ::Debug(string)
      IL_0037:  leave       IL_004d

    }  // end .try
    catch [PLR]PLR.Runtime.ProcessKilledException
    {
      IL_003c:  pop
      IL_003d:  ldarg.0
      IL_003e:  ldstr       "Caught ProcessKilledException"
      IL_0043:  call        instance void [PLR]PLR.Runtime.ProcessBase
                            ::Debug(string)
      IL_0048:  leave       IL_004d

    }  // end handler
    IL_004d:  ldarg.0
    IL_004e:  call        instance void [PLR]PLR.Runtime.ProcessBase::Die()
    IL_0053:  ret
  } // end of method NDC2::RunProcess

} // end of class NDC2
```

B.7 ParallelComposition

```
ParallelComposition = a . 0 | PC2 | 0
PC2 = b . 0
```

are compiled as follows:

```
.class public auto ansi beforefieldinit ParallelComposition
       extends [PLR]PLR.Runtime.ProcessBase
{
  .class auto ansi nested public beforefieldinit Parallel1
         extends [PLR]PLR.Runtime.ProcessBase
  {
    .method public virtual instance void
            RunProcess() cil managed
    {
```

```
    .override [PLR]PLR.Runtime.ProcessBase::RunProcess
    // Code size       84 (0x54)
    .maxstack  10
    .locals init ([0] class [PLR]PLR.Runtime.ChannelSyncAction V_0)
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::InitSetID()
    .try
    {
      IL_0006:  ldarg.0
      IL_0007:  ldstr      "Preparing to sync now..."
      IL_000c:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0011:  ldarg.0
      IL_0012:  ldstr      "a"
      IL_0017:  ldarg.0
      IL_0018:  ldc.i4     0x0
      IL_001d:  ldc.i4.1
      IL_001e:  newobj     instance void [PLR]PLR.Runtime.ChannelSyncAction
                           ::.ctor(string,class
                           [PLR]PLR.Runtime.ProcessBase, int32, bool)
      IL_0023:  stloc.0
      IL_0024:  ldloc.0
      IL_0025:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Sync(class [PLR]PLR.Runtime.IAction)
      IL_002a:  nop
      IL_002b:  nop
      IL_002c:  ldarg.0
      IL_002d:  ldstr      "Turned into 0"
      IL_0032:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0037:  leave      IL_004d

    }  // end .try
    catch [PLR]PLR.Runtime.ProcessKilledException
    {
      IL_003c:  pop
      IL_003d:  ldarg.0
      IL_003e:  ldstr      "Caught ProcessKilledException"
      IL_0043:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0048:  leave      IL_004d

    }  // end handler
    IL_004d:  ldarg.0
    IL_004e:  call       instance void [PLR]PLR.Runtime.ProcessBase::Die()
    IL_0053:  ret
  } // end of method Parallel1::RunProcess

  .method public specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::.ctor()
    IL_0006:  ret
  } // end of method Parallel1::.ctor

} // end of class Parallel1

.class auto ansi nested public beforefieldinit Parallel3
       extends [PLR]PLR.Runtime.ProcessBase
{
  .method public virtual instance void
          RunProcess() cil managed
  {
    .override [PLR]PLR.Runtime.ProcessBase::RunProcess
    // Code size       25 (0x19)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::InitSetID()
    IL_0006:  nop
    IL_0007:  ldarg.0
    IL_0008:  ldstr      "Turned into 0"
    IL_000d:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::Debug(string)
    IL_0012:  ldarg.0
    IL_0013:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::Die()
    IL_0018:  ret
  } // end of method Parallel3::RunProcess

  .method public specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::.ctor()
    IL_0006:  ret
  } // end of method Parallel3::.ctor

} // end of class Parallel3

.method public specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       7 (0x7)
  .maxstack  2
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                       ::.ctor()
  IL_0006:  ret
} // end of method ParallelComposition::.ctor

.method public virtual instance void  RunProcess() cil managed
{
  .override [PLR]PLR.Runtime.ProcessBase::RunProcess
  // Code size       86 (0x56)
  .maxstack  2
  .locals init ([0] class [PLR]PLR.Runtime.ProcessBase V_0,
           [1] class [PLR]PLR.Runtime.ProcessBase V_1,
           [2] class [PLR]PLR.Runtime.ProcessBase V_2)
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase+
                       ::InitSetID()
  IL_0006:  newobj     instance void ParallelComposition/Parallel1
                       ::.ctor()
  IL_000b:  stloc.0
  IL_000c:  ldloc.0
  IL_000d:  ldarg.0
  IL_000e:  call       instance class [PLR]PLR.Runtime.ProcessBase
                       [PLR]PLR.Runtime.ProcessBase::get_Parent()
  IL_0013:  call       instance void [PLR]PLR.Runtime.ProcessBase
                       ::set_Parent(class [PLR]PLR.Runtime.ProcessBase)
  IL_0018:  ldloc.0
  IL_0019:  call       instance void [PLR]PLR.Runtime.ProcessBase::Run()
  IL_001e:  nop
  IL_001f:  newobj     instance void PC2::.ctor()
  IL_0024:  stloc.1
  IL_0025:  ldloc.1
  IL_0026:  ldarg.0
  IL_0027:  call       instance class [PLR]PLR.Runtime.ProcessBase
                       [PLR]PLR.Runtime.ProcessBase::get_Parent()
  IL_002c:  call       instance void [PLR]PLR.Runtime.ProcessBase
                       ::set_Parent(class [PLR]PLR.Runtime.ProcessBase)
  IL_0031:  ldloc.1
  IL_0032:  call       instance void [PLR]PLR.Runtime.ProcessBase::Run()
  IL_0037:  newobj     instance void ParallelComposition/Parallel3::.ctor()
  IL_003c:  stloc.2
  IL_003d:  ldloc.2
  IL_003e:  ldarg.0
  IL_003f:  call       instance class [PLR]PLR.Runtime.ProcessBase
                       [PLR]PLR.Runtime.ProcessBase::get_Parent()
  IL_0044:  call       instance void [PLR]PLR.Runtime.ProcessBase
                       ::set_Parent(class [PLR]PLR.Runtime.ProcessBase)
  IL_0049:  ldloc.2
  IL_004a:  call       instance void [PLR]PLR.Runtime.ProcessBase::Run()
  IL_004f:  ldarg.0
  IL_0050:  call       instance void [PLR]PLR.Runtime.ProcessBase::Die()
  IL_0055:  ret
} // end of method ParallelComposition::RunProcess

} // end of class ParallelComposition

.class public auto ansi beforefieldinit PC2
     extends [PLR]PLR.Runtime.ProcessBase
{
.method public specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       7 (0x7)
  .maxstack  2
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase::.ctor()
  IL_0006:  ret
} // end of method PC2::.ctor

.method public virtual instance void  RunProcess() cil managed
{
  .override [PLR]PLR.Runtime.ProcessBase::RunProcess
  // Code size       84 (0x54)
  .maxstack  10
  .locals init ([0] class [PLR]PLR.Runtime.ChannelSyncAction V_0)
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                       ::InitSetID()
  .try
  {
    IL_0006:  ldarg.0
    IL_0007:  ldstr      "Preparing to sync now..."
    IL_000c:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::Debug(string)
    IL_0011:  ldarg.0
    IL_0012:  ldstr      "b"
    IL_0017:  ldarg.0
    IL_0018:  ldc.i4     0x0
    IL_001d:  ldc.i4.1
    IL_001e:  newobj     instance void [PLR]PLR.Runtime.ChannelSyncAction
                         ::.ctor(string,
                         class [PLR]PLR.Runtime.ProcessBase, int32, bool)
    IL_0023:  stloc.0
    IL_0024:  ldloc.0
    IL_0025:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::Sync(class [PLR]PLR.Runtime.IAction)
    IL_002a:  nop
    IL_002b:  nop
    IL_002c:  ldarg.0
    IL_002d:  ldstr      "Turned into 0"
    IL_0032:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::Debug(string)
    IL_0037:  leave      IL_004d

  }  // end .try
  catch [PLR]PLR.Runtime.ProcessKilledException
  {
    IL_003c:  pop
    IL_003d:  ldarg.0
    IL_003e:  ldstr      "Caught ProcessKilledException"
    IL_0043:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::Debug(string)
    IL_0048:  leave      IL_004d

  }  // end handler
  IL_004d:  ldarg.0
  IL_004e:  call       instance void [PLR]PLR.Runtime.ProcessBase::Die()
  IL_0053:  ret
} // end of method PC2::RunProcess

} // end of class PC2
```

B.8 MethodCall

```
MethodCall = :Print("Hello") . 0
```

is compiled as follows:

```
.class public auto ansi beforefieldinit MethodCall
       extends [PLR]PLR.Runtime.ProcessBase
{
  .method public specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase::.ctor()
    IL_0006:  ret
  } // end of method MethodCall::.ctor

  .method public virtual instance void  RunProcess() cil managed
  {
    .override [PLR]PLR.Runtime.ProcessBase::RunProcess
    // Code size       86 (0x56)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::InitSetID()
    .try
    {
      IL_0006:  ldarg.0
      IL_0007:  ldstr      "Preparing to sync now..."
      IL_000c:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0011:  ldarg.0
      IL_0012:  ldstr      "Print(\"Hello\")"
      IL_0017:  ldarg.0
      IL_0018:  newobj     instance void [PLR]PLR.Runtime.MethodCallAction
                           ::.ctor(string,
                           class [PLR]PLR.Runtime.ProcessBase)
      IL_001d:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Sync(class [PLR]PLR.Runtime.IAction)
      IL_0022:  nop
      IL_0023:  ldstr      "Hello"
      IL_0028:  call       void [PLR]PLR.Runtime.BuiltIns::Print(object)
      IL_002d:  nop
      IL_002e:  ldarg.0
      IL_002f:  ldstr      "Turned into 0"
      IL_0034:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0039:  leave      IL_004f

    }  // end .try
    catch [PLR]PLR.Runtime.ProcessKilledException
    {
      IL_003e:  pop
      IL_003f:  ldarg.0
      IL_0040:  ldstr      "Caught ProcessKilledException"
      IL_0045:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_004a:  leave      IL_004f

    }  // end handler
    IL_004f:  ldarg.0
    IL_0050:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::Die()
    IL_0055:  ret
  } // end of method MethodCall::RunProcess

} // end of class MethodCall
```

B.9 Restrict

```
  Restrict = ( a . (d . 0)\ d ) \{a}
```

is compiled as follows:

```
.class public auto ansi beforefieldinit Restrict
       extends [PLR]PLR.Runtime.ProcessBase
{
  .class auto ansi nested public beforefieldinit Inner
```

```
       extends [PLR]PLR.Runtime.ProcessBase
{
  .method public static bool  RestrictByName(
      class [PLR]PLR.Runtime.IAction A_0) cil managed
  {
    // Code size       37 (0x25)
    .maxstack  3
    .locals init ([0] class [PLR]PLR.Runtime.ChannelSyncAction V_0)
    IL_0000:  ldarg.0
    IL_0001:  isinst     [PLR]PLR.Runtime.ChannelSyncAction
    IL_0006:  brtrue     IL_000d

    IL_000b:  ldc.i4.0
    IL_000c:  ret

    IL_000d:  ldarg.0
    IL_000e:  castclass  [PLR]PLR.Runtime.ChannelSyncAction
    IL_0013:  stloc.0
    IL_0014:  ldloc.0
    IL_0015:  call       instance string [PLR]PLR.Runtime.ChannelSyncAction
                         ::get_Name()
    IL_001a:  ldstr      "d"
    IL_001f:  call       bool [mscorlib]System.String::op_Equality(string,
                                                                    string)
    IL_0024:  ret
  } // end of method Inner::RestrictByName

  .method public virtual instance class [PLR]PLR.Runtime.RestrictAction
        get_Restrict() cil managed
  {
    .override [PLR]PLR.Runtime.ProcessBase::get_Restrict
    // Code size       13 (0xd)
    .maxstack  3
    IL_0000:  ldnull
    IL_0001:  ldftn      bool Restrict/Inner::RestrictByName(class
                         [PLR]PLR.Runtime.IAction)
    IL_0007:  newobj     instance void [PLR]PLR.Runtime.RestrictAction
                         ::.ctor(object, native int)
    IL_000c:  ret
  } // end of method Inner::get_Restrict

  .method public virtual instance void
        RunProcess() cil managed
  {
    .override [PLR]PLR.Runtime.ProcessBase::RunProcess
    // Code size       84 (0x54)
    .maxstack  10
    .locals init ([0] class [PLR]PLR.Runtime.ChannelSyncAction V_0)
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::InitSetID()
    .try
    {
      IL_0006:  ldarg.0
      IL_0007:  ldstr      "Preparing to sync now..."
      IL_000c:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0011:  ldarg.0
      IL_0012:  ldstr      "d"
      IL_0017:  ldarg.0
      IL_0018:  ldc.i4     0x0
      IL_001d:  ldc.i4.1
      IL_001e:  newobj     instance void [PLR]PLR.Runtime.ChannelSyncAction
                           ::.ctor(string, class
                           [PLR]PLR.Runtime.ProcessBase, int32, bool)
      IL_0023:  stloc.0
      IL_0024:  ldloc.0
      IL_0025:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Sync(class [PLR]PLR.Runtime.IAction)
      IL_002a:  nop
      IL_002b:  nop
      IL_002c:  ldarg.0
      IL_002d:  ldstr      "Turned into 0"
      IL_0032:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0037:  leave      IL_004d

    }  // end .try
    catch [PLR]PLR.Runtime.ProcessKilledException
    {
      IL_003c:  pop
      IL_003d:  ldarg.0
      IL_003e:  ldstr      "Caught ProcessKilledException"
      IL_0043:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0048:  leave      IL_004d

    }  // end handler
    IL_004d:  ldarg.0
    IL_004e:  call       instance void [PLR]PLR.Runtime.ProcessBase::Die()
    IL_0053:  ret
  } // end of method Inner::RunProcess

  .method public specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::.ctor()
    IL_0006:  ret
  } // end of method Inner::.ctor

} // end of class Inner

.method public specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       7 (0x7)
  .maxstack  2
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                       ::.ctor()
  IL_0006:  ret
} // end of method Restrict::.ctor

.method public static bool  RestrictByName(class [PLR]PLR.Runtime.IAction
    A_0) cil managed
{
  // Code size       37 (0x25)
  .maxstack  3
  .locals init ([0] class [PLR]PLR.Runtime.ChannelSyncAction V_0)
  IL_0000:  ldarg.0
  IL_0001:  isinst     [PLR]PLR.Runtime.ChannelSyncAction
  IL_0006:  brtrue     IL_000d

  IL_000b:  ldc.i4.0
  IL_000c:  ret
```

```
    IL_000d:  ldarg.0
    IL_000e:  castclass  [PLR]PLR.Runtime.ChannelSyncAction
    IL_0013:  stloc.0
    IL_0014:  ldloc.0
    IL_0015:  call       instance string [PLR]PLR.Runtime.ChannelSyncAction
                         ::get_Name()
    IL_001a:  ldstr      "a"
    IL_001f:  call       bool [mscorlib]System.String::op_Equality(string,
                                                                   string)
    IL_0024:  ret
  } // end of method Restrict::RestrictByName

  .method public virtual instance class [PLR]PLR.Runtime.RestrictAction
          get_Restrict() cil managed
  {
    .override [PLR]PLR.Runtime.ProcessBase::get_Restrict
    // Code size       13 (0xd)
    .maxstack  3
    IL_0000:  ldnull
    IL_0001:  ldftn      bool Restrict::RestrictByName(class
                         [PLR]PLR.Runtime.IAction)
    IL_0007:  newobj     instance void [PLR]PLR.Runtime.RestrictAction
                         ::.ctor(object, native int)
    IL_000c:  ret
  } // end of method Restrict::get_Restrict

  .method public virtual instance void  RunProcess() cil managed
  {
    .override [PLR]PLR.Runtime.ProcessBase::RunProcess
    // Code size       103 (0x67)
    .maxstack  10
    .locals init ([0] class [PLR]PLR.Runtime.ChannelSyncAction V_0,
             [1] class [PLR]PLR.Runtime.ProcessBase V_1)
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::InitSetID()
    .try
    {
      IL_0006:  ldarg.0
      IL_0007:  ldstr      "Preparing to sync now..."
      IL_000c:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_0011:  ldarg.0
      IL_0012:  ldstr      "a"
      IL_0017:  ldarg.0
      IL_0018:  ldc.i4     0x0
      IL_001d:  ldc.i4.1
      IL_001e:  newobj     instance void [PLR]PLR.Runtime.ChannelSyncAction
                           ::.ctor(string, class [PLR]PLR.Runtime.ProcessBase
                           , int32, bool)
      IL_0023:  stloc.0
      IL_0024:  ldloc.0
      IL_0025:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Sync(class [PLR]PLR.Runtime.IAction)
      IL_002a:  nop
      IL_002b:  newobj     instance void Restrict/Inner::.ctor()
      IL_0030:  stloc.1
      IL_0031:  ldloc.1
      IL_0032:  ldarg.0
      IL_0033:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::set_Parent(class [PLR]PLR.Runtime.ProcessBase)
      IL_0038:  ldloc.1
      IL_0039:  ldarg.0
      IL_003a:  call       instance valuetype [mscorlib]System.Guid
                           [PLR]PLR.Runtime.ProcessBase::get_SetID()
      IL_003f:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::set_SetID(valuetype [mscorlib]System.Guid)
      IL_0044:  ldloc.1
      IL_0045:  call       instance void [PLR]PLR.Runtime.ProcessBase::Run()
      IL_004a:  leave      IL_0060

    }  // end .try
    catch [PLR]PLR.Runtime.ProcessKilledException
    {
      IL_004f:  pop
      IL_0050:  ldarg.0
      IL_0051:  ldstr      "Caught ProcessKilledException"
      IL_0056:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::Debug(string)
      IL_005b:  leave      IL_0060

    }  // end handler
    IL_0060:  ldarg.0
    IL_0061:  call       instance void [PLR]PLR.Runtime.ProcessBase::Die()
    IL_0066:  ret
  } // end of method Restrict::RunProcess

} // end of class Restrict
```

B.10 Relabel

```
Relabel = ( a . (d . 0)[dnew/d] )[anew/a]
```

is compiled as follows:

```
.class public auto ansi beforefieldinit Relabel
       extends [PLR]PLR.Runtime.ProcessBase
{
  .class auto ansi nested public beforefieldinit Inner
         extends [PLR]PLR.Runtime.ProcessBase
  {
    .method public static class [PLR]PLR.Runtime.IAction
            RelabelAction(class [PLR]PLR.Runtime.IAction A_0) cil managed
    {
      // Code size       59 (0x3b)
      .maxstack  4
      .locals init ([0] class [PLR]PLR.Runtime.ChannelSyncAction V_0)
      IL_0000:  ldarg.0
      IL_0001:  isinst     [PLR]PLR.Runtime.ChannelSyncAction
      IL_0006:  brtrue     IL_000d

      IL_000b:  ldarg.0
      IL_000c:  ret

      IL_000d:  ldarg.0
      IL_000e:  castclass  [PLR]PLR.Runtime.ChannelSyncAction
      IL_0013:  stloc.0
      IL_0014:  ldloc.0
      IL_0015:  call       instance string [PLR]PLR.Runtime.ChannelSyncAction
                           ::get_Name()
      IL_001a:  ldstr       "d"
      IL_001f:  call        bool [mscorlib]System.String::op_Equality(string,
                                                                     string)
      IL_0024:  brfalse     IL_0039

      IL_0029:  ldloc.0
      IL_002a:  ldstr       "dnew"
      IL_002f:  call        instance void [PLR]PLR.Runtime.ChannelSyncAction
                            ::set_Name(string)
      IL_0034:  br          IL_0039

      IL_0039:  ldloc.0
      IL_003a:  ret
    } // end of method Inner::RelabelAction

    .method public virtual instance class [PLR]PLR.Runtime.PreProcessAction
            get_PreProcess() cil managed
    {
      .override [PLR]PLR.Runtime.ProcessBase::get_PreProcess
      // Code size       13 (0xd)
      .maxstack  3
      IL_0000:  ldnull
      IL_0001:  ldftn       class [PLR]PLR.Runtime.IAction Relabel/Inner
                            ::RelabelAction(class [PLR]PLR.Runtime.IAction)
      IL_0007:  newobj      instance void [PLR]PLR.Runtime.PreProcessAction
                            ::.ctor(object, native int)
      IL_000c:  ret
    } // end of method Inner::get_PreProcess

    .method public virtual instance void
            RunProcess() cil managed
    {
      .override [PLR]PLR.Runtime.ProcessBase::RunProcess
      // Code size       84 (0x54)
      .maxstack  10
      .locals init ([0] class [PLR]PLR.Runtime.ChannelSyncAction V_0)
      IL_0000:  ldarg.0
      IL_0001:  call        instance void [PLR]PLR.Runtime.ProcessBase
                            ::InitSetID()
      .try
      {
        IL_0006:  ldarg.0
        IL_0007:  ldstr       "Preparing to sync now..."
        IL_000c:  call        instance void [PLR]PLR.Runtime.ProcessBase
                              ::Debug(string)
        IL_0011:  ldarg.0
        IL_0012:  ldstr       "d"
        IL_0017:  ldarg.0
        IL_0018:  ldc.i4      0x0
        IL_001d:  ldc.i4.1
        IL_001e:  newobj      instance void [PLR]PLR.Runtime.ChannelSyncAction
                              ::.ctor(string, class
                              [PLR]PLR.Runtime.ProcessBase, int32, bool)
        IL_0023:  stloc.0
        IL_0024:  ldloc.0
        IL_0025:  call        instance void [PLR]PLR.Runtime.ProcessBase
                              ::Sync(class [PLR]PLR.Runtime.IAction)
        IL_002a:  nop
        IL_002b:  nop
        IL_002c:  ldarg.0
        IL_002d:  ldstr       "Turned into 0"
        IL_0032:  call        instance void [PLR]PLR.Runtime.ProcessBase
                              ::Debug(string)
        IL_0037:  leave       IL_004d
      }  // end .try
      catch [PLR]PLR.Runtime.ProcessKilledException
      {
        IL_003c:  pop
        IL_003d:  ldarg.0
        IL_003e:  ldstr      "Caught ProcessKilledException"
        IL_0043:  call       instance void [PLR]PLR.Runtime.ProcessBase
                             ::Debug(string)
        IL_0048:  leave      IL_004d

      }  // end handler
      IL_004d:  ldarg.0
      IL_004e:  call       instance void [PLR]PLR.Runtime.ProcessBase::Die()
      IL_0053:  ret
    } // end of method Inner::RunProcess

    .method public specialname rtspecialname
            instance void  .ctor() cil managed
    {
      // Code size       7 (0x7)
      .maxstack  2
      IL_0000:  ldarg.0
      IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                           ::.ctor()
      IL_0006:  ret
    } // end of method Inner::.ctor

  } // end of class Inner

  .method public specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  2
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [PLR]PLR.Runtime.ProcessBase
                         ::.ctor()
    IL_0006:  ret
  } // end of method Relabel::.ctor

  .method public static class [PLR]PLR.Runtime.IAction
          RelabelAction(class [PLR]PLR.Runtime.IAction A_0) cil managed
  {
    // Code size       59 (0x3b)
    .maxstack  4
    .locals init ([0] class [PLR]PLR.Runtime.ChannelSyncAction V_0)
    IL_0000:  ldarg.0
    IL_0001:  isinst     [PLR]PLR.Runtime.ChannelSyncAction
    IL_0006:  brtrue     IL_000d

    IL_000b:  ldarg.0
    IL_000c:  ret

    IL_000d:  ldarg.0
    IL_000e:  castclass  [PLR]PLR.Runtime.ChannelSyncAction
    IL_0013:  stloc.0
    IL_0014:  ldloc.0
    IL_0015:  call       instance string [PLR]PLR.Runtime.ChannelSyncAction
                         ::get_Name()
    IL_001a:  ldstr      "a"
    IL_001f:  call       bool [mscorlib]System.String::op_Equality(string,
                                                                 string)
    IL_0024:  brfalse    IL_0039

    IL_0029:  ldloc.0
    IL_002a:  ldstr       "anew"
    IL_002f:  call        instance void [PLR]PLR.Runtime.ChannelSyncAction
                          ::set_Name(string)
    IL_0034:  br          IL_0039

    IL_0039:  ldloc.0
    IL_003a:  ret
  } // end of method Relabel::RelabelAction

  .method public virtual instance class [PLR]PLR.Runtime.PreProcessAction
          get_PreProcess() cil managed
  {
    .override [PLR]PLR.Runtime.ProcessBase::get_PreProcess
    // Code size       13 (0xd)
    .maxstack  3
    IL_0000:  ldnull
    IL_0001:  ldftn       class [PLR]PLR.Runtime.IAction Relabel
                          ::RelabelAction(class [PLR]PLR.Runtime.IAction)
    IL_0007:  newobj      instance void [PLR]PLR.Runtime.PreProcessAction
                          ::.ctor(object, native int)
    IL_000c:  ret
  } // end of method Relabel::get_PreProcess

  .method public virtual instance void  RunProcess() cil managed
  {
    .override [PLR]PLR.Runtime.ProcessBase::RunProcess
    // Code size       103 (0x67)
    .maxstack  10
    .locals init ([0] class [PLR]PLR.Runtime.ChannelSyncAction V_0,
             [1] class [PLR]PLR.Runtime.ProcessBase V_1)
    IL_0000:  ldarg.0
    IL_0001:  call        instance void [PLR]PLR.Runtime.ProcessBase
                          ::InitSetID()
    .try
    {
      IL_0006:  ldarg.0
      IL_0007:  ldstr       "Preparing to sync now..."
      IL_000c:  call        instance void [PLR]PLR.Runtime.ProcessBase
                            ::Debug(string)
      IL_0011:  ldarg.0
      IL_0012:  ldstr       "a"
      IL_0017:  ldarg.0
      IL_0018:  ldc.i4      0x0
      IL_001d:  ldc.i4.1
      IL_001e:  newobj      instance void [PLR]PLR.Runtime.ChannelSyncAction
                            ::.ctor(string, class
                            [PLR]PLR.Runtime.ProcessBase, int32, bool)
      IL_0023:  stloc.0
      IL_0024:  ldloc.0
      IL_0025:  call        instance void [PLR]PLR.Runtime.ProcessBase
                            ::Sync(class [PLR]PLR.Runtime.IAction)
      IL_002a:  nop
      IL_002b:  newobj      instance void Relabel/Inner::.ctor()
      IL_0030:  stloc.1
      IL_0031:  ldloc.1
      IL_0032:  ldarg.0
      IL_0033:  call        instance void [PLR]PLR.Runtime.ProcessBase
                            ::set_Parent(class [PLR]PLR.Runtime.ProcessBase)
      IL_0038:  ldloc.1
      IL_0039:  ldarg.0
      IL_003a:  call        instance valuetype [mscorlib]System.Guid
                            [PLR]PLR.Runtime.ProcessBase::get_SetID()
      IL_003f:  call        instance void [PLR]PLR.Runtime.ProcessBase
                            ::set_SetID(valuetype [mscorlib]System.Guid)
      IL_0044:  ldloc.1
      IL_0045:  call        instance void [PLR]PLR.Runtime.ProcessBase::Run()
      IL_004a:  leave       IL_0060

    }  // end .try
    catch [PLR]PLR.Runtime.ProcessKilledException
    {
      IL_004f:  pop
      IL_0050:  ldarg.0
      IL_0051:  ldstr       "Caught ProcessKilledException"
      IL_0056:  call        instance void [PLR]PLR.Runtime.ProcessBase
                            ::Debug(string)
      IL_005b:  leave       IL_0060

    }  // end handler
    IL_0060:  ldarg.0
    IL_0061:  call        instance void [PLR]PLR.Runtime.ProcessBase::Die()
    IL_0066:  ret
  } // end of method Relabel::RunProcess

} // end of class Relabel
```

Bibliography

[1] Luca Aceto, Anna Ingólfsdóttir, Kim G. Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification.* Cambridge University Press, 2007.

[2] Giordano Battilana. *Virtual Machines for Aspect-Oriented Systems.* Technical University of Denmark, 2008.

[3] Lorenzo Bettini, Rocco De Nicola, and Rosario Pugliese. *X-Klaim and Klava: Programming Mobile Code*. In M. Lenisa and M. Miculan, editors, *TOSCA 2001*, volume 62. Elsevier, 2001.

[4] Lorenzo Bettini, Rocco De Nicola, and Rosario Pugliese. Klava: a java package for distributed and mobile applications. *Softw. Pract. Exper.*, 32(14):1365–1394, 2002.

[5] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[6] Microsoft Corporation. *Compiling MSIL to Native Code.* http://msdn.microsoft.com/en-us/library/ht8ecch6.aspx [Online; accessed 04-02-2009].

[7] Microsoft Corporation. *Dynamic Language Runtime.* http://www.codeplex.com/dlr [Online; accessed 12-01-2009].

[8] Microsoft Corporation. *Managed Babel.* http://msdn.microsoft.com/en-us/library/bb165037.aspx [Online; accessed 02-05-2009].

[9] Peter Van Eijk. *Systematically Constructing a CCS Interpreter - Summary*, 1988. http://en.scientificcommons.org/42999072[Online; accessed 14-02-2009].

[10] Leonardo Freitas. JACK: A process algebra implementation in Java. Master's thesis, Centro de Informatica, Universidade Federal de Pernambuco, April 2002. http://www.cin.ufpe.br/ lf25.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[12] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[13] John Gough and Wayne Kelly. *The GPPG Parser Generator*. http://plas.fit.qut.edu.au/gppg/files/gppg.pdf [Online; accessed 02-05-2009].

[14] Chris Hankin, Flemming Nielson, Hanne Riis Nielson, and Fan Yang. Advice for coordination. In *COORDINATION*, pages 153–168, 2008.

[15] Ed Harcourt. *An Extensible Interpreter for Value-Passing CCS*, 1995.

[16] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[17] Christian Holm, Mike Kruger, and Bernhard Spuida. *Dissecting a C# Application: Inside SharpDevelop*. Wrox Press, January 2003.

[18] John Levine, Tony Mason, and Doug Brown. *lex & yacc, 2nd Edition (A Nutshell Handbook)*. O'Reilly, October 1992.

[19] George Milne and Robin Milner. Concurrent processes and their syntax. *J. ACM*, 26(2):302–321, 1979.

[20] R. Milner. An approach to the semantics of parallel programs. In *Proceedings Convegno di Informatica Teorica, Pisa*, pages 283–302, 1973.

[21] R. Milner. Algebras for communicating systems. In *Proc. AFCET/SMF joint colloq. Applied Mathematics, Paris*, 1978.

[22] R. Milner. *A Calculus of Communicating Systems.*, volume 92 of *LNCS*. Springer Verlag, 1980.

[23] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[24] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part i. *I and II. Information and Computation*, 100, 1989.

[25] Hanspeter Mössenböck and Johannes Kepler. *The Compiler Generator Coco/R.* http://www.ssw.uni-linz.ac.at/coco/Doc/UserManual.pdf [Online; accessed 15-01-2009].

[26] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. Klaim: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[27] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, December 2004.

[28] Thomas Noll and Chanchal Kumar Roy. Modeling erlang in the pi-calculus. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 72–77, New York, NY, USA, 2005. ACM.

[29] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*, pages 9–15, 19–21 1998.

[30] P.H. Welch and F.R.M. Barnes. Communicating mobile processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005. to appear.