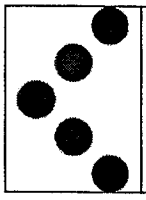

PORTABLE PARALLEL COMPUTING BASED ON VIRTUAL MACHINES

Tool for combining re-usable aspects - PSL

Eduardo Augusto Peixoto da Silva Brito

Supervisor: João Luís Sobral

February 2008

1. INTRODUCTION **2**

2. MOTIVATION **3**

3. PSL **4**

3.1 DEFINITION OF TERMS 4
3.2 DESIGN OVERVIEW 5
3.3 LANGUAGES 5
3.3.1 SPECIFICATION FILE (MAKEGENERICS) 6
3.3.1.1 Environment Variables *6*
3.3.1.2 Built-in Templates *6*
3.3.1.3 Calling external template files *8*
3.3.1.4 Composition and precedence *8*
3.3.1.4.1 Explicit chaining *8*
3.3.1.4.2 Implicit chaining *8*
3.3.1.5 Understanding specification files *9*
3.3.2 GENERIC TEMPLATE FILES 11
3.3.2.1 Pure Templates *12*
3.3.2.2 Compositional Templates *13*
3.3.2.3 Reserved keywords for Template tags *14*
3.3.2.4 Parameters *15*
3.3.2.5 Execution block *15*
3.3.2.6 Understanding template files *16*
3.3.7 Possible improvements to the design *19*

3.4 IMPLEMENTATION **20**

3.4.1 USED TOOLS 20
3.4.2 INSTALLATION AND RUNNING 20
3.4.3 ACHIEVEMENTS 20
3.4.4 INITIAL PROBLEMS 21
3.4.5 BRIEF OVERVIEW AND EXPLANATION OF THE CLASSES 22
3.4.6 WHAT CAN BE IMPROVED? 23

APPENDIX **25**

A – GRAMMARS 25
A.1 - Specification Files Parser *25*
A.2 – Specification Files Tree Parser *27*
A.3 – Specification File Lexer *29*
A.4 – Template Files Parser *31*
A.5 – Template Files Tree Parser *33*
A.6 – Tempate Files Lexer *35*
B – SYNTAX HANDBOOK 37
B.1 – Specification file *37*

1. Introduction

The PSL (Parallelism Specification Language) was developed from a project at the Universidade of Minho where there was need for a tool that could, given a specification, generate code to parallelise and distribute Java applications using some well known design patterns.

The work that is presented here was done under the supervision of João Luís Ferreira Sobral in the context of the PPC-VM project.

2. Motivation

When a programmer wants to introduce code to parallelise (and/or distribute) his sequential code (may it be Java or not) he often does not have a specification for it and he is also forced to modify and add extra complexity to existing and working code. Also, most of the time, programmers have to (re-)code the same patterns several times because there is no library or tool that can be (re-)used and adapted to work with the existing code or, at least, there is not a library that is able to handle complex specifications of multi-threaded and distributed applications. Another problem that exists when creating code for a concurrent/parallel/distributed setting is that these applications are of a non-deterministic nature and, most of the times, bugs are hard to find, replicate and fix.

The PSL solves this problems by specifying in a separate text file how the application is going to be parallelised and/or distributed; this is done by using existing templates that implement (in AspectJ) some well known design patterns and also by allowing for the inclusion of new templates created by the programmer. Using the existing templates of the PSL, the development of this kind of applications is much faster and safer because the PSL automatically generates code for the given specification and it also pays attention to how the templates interact with each other using an well defined composition model to ensure that the behaviour is as it is supposed to be (and this is based on the way that the templates are "glued" together on the specification file).

3. PSL

3.1 Definition of terms

Before explaining the PSL in detail, it is best to clarify some terms that will be used throughout the rest of this report.

- **Templates:** These are the source code files that implement the design patterns that are used by the PSL. They are made mainly of AspectJ code but they obey a specific syntax (that will be showed here in the report) and can contain a certain number of ***tags*** that are reserved for the internal use of the tool.

- **Tags**: Used inside the template files, these tags are a way of passing specific parameters onto the PSL; some are generic and created by the person that implemented the template but others have a very specific meaning to the PSL like the MAIN tag or the ASPECT tag (both of which will be discussed upfront). These tags are identified by their tag name and by being surrounded by "<" and ">".

- **Composition**: In this context it is used to specify how several aspects are connected with each other. An aspect is connected with another if it uses or is used by it.

- **Specification file/MakeGenerics**: This is the file that serves as input to the PSL; it is dependent of the implementation of the PSL and can be changed if wished (the reasoning behind this was to follow the same pattern as make/Makefile).

3.2 Design overview

The PSL was designed to be used as a "CAD tool" in which the programmer would specify how the application would be parallelised and distributed by using the templates that were given to him or made by him. This was needed because, with the previous approach (Java Annotations) there were problems when several design patterns were used (and even when the same pattern was used several times) in the same source file, because they would not combine well and the application did not work as it was planned to work. At a later stage it was created a mechanism that made possible the inclusion of new templates and with that it was possible for the PSL to be extended at will.

The design of the grammar was heavily influenced by the technology that is being used (Java and AspectJ); it could not be too much abstract because of the nature of AspectJ. Some things had to be implemented and be expressed in the syntax of the PSL so that it would generate code (automatically) that was correct and performed as it was intended to perform.

One of the main problems of AspectJ is that, to make the aspects work with a java application, it needs to be in the same directory as the source code and, when there are several aspects in the same directory acting upon the same code and each others, they often clash and generate errors that would not exist if they were "single" aspects. There is also clear advantages in using AspectJ and AOP for doing what we intended to do (by preventing tangled code, separating between the original source code and the additional features, etc).

Another objective that we tried to achieve with the PSL was to have the generated code as close as possible to what a programmer would write in the same situation. That is, in a way, what actually happens naturally; someone creates the templates and then the PSL uses that to generate the code for the given specification. The advantage in using the PSL is that it controls the composition, automatically, and by doing this it reduces the development time that is needed to do this manually.

The PSL is also very small and powerful and it allows for complex specifications to be described in few lines of code; it also as a very nice and small learning curve.

3.3 Languages

The proper syntax in BNF notation is supplied as an appendix to the report. The next sections will show the particularities of the languages, the options that have been made while designing it and its syntax, in an informal way.

3.3.1 *Specification file (MakeGenerics)*

As the name implies, this is the file where we describe how the patterns are going to compose with each other.

The structure for this file is:

```
programName {

(...) #statements

}
```

3.3.1.1 Environment Variables

There are only two:

- **Main**: It is used to identify what is the main method that will be used by the program. Some aspects may need to know this.
- **Package**: This defines the package where the files will be put. Although this has been deprecated it can still be used but it will only influence the aspects that have not defined their package when they were declared. It can be useful when we want several aspects to be in the same package.

3.3.1.2 Built-in Templates

The PSL has some built-in/hard-coded templates. This was needed in the beginning of the PSL, before there was a possibility of creating and adding generic templates; they were maintained because there was no need to re-implement the templates even though the PSL is now powerful enough to do that.

What is hard-coded is the syntax and the generation algorithm for the template; the template itself is coded in a single separate archive.

The templates that are available are:

- **Replicate**: Intercepts the creation of an object and generates a given number of clones.
- **Delegate**: Randomly attributes the intercepted method to a cloned object. It is optionally used by the replicate template.
- **Broadcast**: Sends the same message to all the objects, including the original. It is optionally used by the replicate template.
- **Scatter**: It is the same as the broadcast method but it uses a given method to send messages with different parameters to each object. It is optionally used by the replicate template.

  - **Reduce**: It is an optional parameter of the scatter template; it collects the results of the scatter computation
- **After and Before**: They simulate the same behaviour as their AspectJ counterparts; they are to be used very carefully and not often. It is optionally used by the replicate template.

Replicate is the only that can be invoked alone; all the other templates are called from within the Replicate; this was needed in the early stages of the PSL to set dependencies because all of them needed the copies of the object that the replicate creates.

The full syntax of these templates is:

- **Replicate< Aspect, Method, Class|Replicate, [List of Templates,] Number>**
  - **Aspect**: The aspect that will be used for the generated replicate
  - **Method**: Specifies what is the method where the objects are going to be created
  - **Class|Replicate**: The class that is going to be replicated; it can also be another replicate.
  - **List of Templates**: Optional list of templates that will be used with this replicate.
  - **Number**: Number of clones to be created. It is a number and not a string.
- **Delegate<Pointcut>**
  - **Pointcut**: The intercepted method that will be randomly directed to a clone
- **Broadcast<Pointcut[, Reduce<Function>]>**
  - **Pointcut**: Same as above
  - **Reduce<Function>**
    - The Reduce template. It is optional.
      - **Function**: is used to specify how the results are to be gathered; it is only needed the function name. This function can be created inside the PSL using Method or can be called using a statically defined method. It takes an ArrayList as argument.
- **Scatter<Pointcut[, Reduce<Function>]>**
  - **Pointcut**: Same as above.
    - **Reduce<Function>**: The same as with broadcast
- **After/Before<Pointcut, Function>**
  - **Pointcut**: Same as above
  - **Function**: The function that is called after/before the pointcut is intercepted

3.3.1.3 Calling external template files

The PSL is also capable of calling generic templates (that are written in a specific language). To call this templates there are a few things that have to be met.

- The template must be in the Templates directory of the compiler (implementation wise)
- The first character has to be upper case and the extension has to be .tpl
- It has to be written in a specific language

After doing that, the template can be called from a specification file by invoking the template using the NameOfTemplate<parameter1, ..., parameterN>; this is similar to calling an internal template.

The writing of external template files will be detailed further in this document.

3.3.1.4 Composition and precedence

The composition is handled by looking at the implicit and explicit chaining of the templates.

3.3.1.4.1 Explicit chaining

This is the most important method of composition. It combines the templates by looking at their dependencies. This method checks for templates that are using other templates as a parameter (as an example, the replicate can have other replicates inside of them; it is the third parameter of a replicate). This defines a natural order for the composition of the parameters. It is also normal for a generic template to have connections to other templates (this is usually the second parameter of a generic template, being that the first parameter is usually the filename to be used by the generator).

3.3.1.4.2 Implicit chaining

Some templates are defined without the need to chain with other templates. When this is the case, these templates should have the least priority (or the most priority; depends on the implementation but it really should not matter, whatever the case is).

3.3.1.5 Understanding specification files

In this section we will show and explain some simple specification files.

```
testFilter {

#Basic test for Filter

      Aspect asp1 = <'Dist'>;
      cls = 'CryptBench';
      pcut = 'JGFinitialise()';
      pcut2 = 'Do()';

      Replicate<asp1, 'Filter.main', cls, 2,
                Scatter<pcut, iscatterFunc.scatter'>,
                Broadcast<pcut2, Reduce< iscatterFunc.reduce'>>
            >;
}
```

- **testFilter** is the name of the aspect; it is only used for clarity and it is not important. This starts a block of code for the specification file.
- **#Basic test for Filter** is a comment saying what the file is supposed to do. Comments start with the # character.
- **Aspect asp1 = <'Dist'>**
  - **Aspect asp1** = <'Dist'> is saying that there will be an aspect, identified by asp1, that will have the name Dist.
  - **'Dist'** is a string.
- **cls**, **pcut** and **pcut2**, are variables used only to store 3 strings. They are used to simplify the writing of the file.
- **Replicate**<asp1, 'Filter.main', cls, 2,**Scatter**<pcut, iscatterFunc.scatter'>, **Broadcast**<pcut2, **Reduce**< iscatterFunc.reduce'>> >;
  - **Replicate<asp1, 'Filter.main', cls, 2,** ... means that it will store the generated aspect in asp1 (Dist.aj). The aspect (Replicate) will intercept the creation of objects of the type cls (CryptBench) inside of Filter.main and it will make 2 copies of them.
  - **Scatter<pcut, 'scatterFunc.scatter'>** will intercept the execution of the method pcut (JGFInitialise) from cls (CryptBench) and will do a scatter to the copies created by replicate using a statically defined method defined as ScatterFunc.scatter to modify the parameters; this is a method supplied by the programmer and it is not from the PSL.
  - **Broadcast<pcut2, Reduce<'scatterFunc.reduce'>>** will intercept calls to pcut2 (DO), broadcast the method to the copies created by replicate and will perform a reduce using the method ScatterFunc.reduce; this method is supplied by the programmer and it is not from the PSL.

```
example2{

        MAIN = 'MandelClient.main';
        Aspect asp1 = <'Teste1'>;
        Aspect asp2 = <'Teste2'>;
        Aspect rep = <'RepTeste'>;

        cls = 'MandServer';
        pcut = ishort[][] mandelBrot(Complex c0, double gap, int x0, int y0, int
                width, int height, int nit)';
        pcut2 = 'void funteste()';

        Replicate<rep, MAIN, cls, 2>;

#External template file
        TestComposition<asp1, asp2, MAIN, cls, pcut, pcut2, iseparators.sep'>;
```

The two most important things in this example are MAIN and TestComposition. MAIN is using an ambient variable and it is setting it to MandelClient.main. All aspects that use this variable will be bind to that value.

TestComposition is an example of how to call external template files. For it to be possible, you should follow the guidelines on how to create new template files and give the correct input parameters. What is important to note here is that the invocation of an external or internal template is done in the same way.

3.3.2 Generic Template Files

These files can be added to the PSL to extend the number of patterns that can be used inside the specification. There are two types of template files: Pure Template Files and Compositional Template Files. They both have this generic structure:

```
Parameters = [PAR1, PAR2, ..., PAR3];

Execution = BlockName<Returns, Void>; (Optional)

Template / Composition {

(...) #Specific information regarding each type

}
```

We use the Pure Templates when we want to code a new pattern to be added to the PSL that is independent of all other templates. It is a completely new template that is self-contained. Every external template creates a separate file (the built-in templates are not forced to always do this).

The Compositional Templates are used to link several templates into a new template; we can link together, for example, a Replicate, a Separate and a Broadcast and call it a Distribution template.

3.3.2.1 Pure Templates

The specific (informal) structure of these templates is:

```
Parameters = [PAR1, PAR2, ..., PAR3];

Execution = ExecutionName<Returns, Void>; (optional)

Template {

        CORE = 'Core AspectJ/Java application code';

        BLOCKa = 'These are...';
        (....)
        BLOCKn = '...Helper blocks';
}
```

This defines a Pure Template. We use the Parameters variable to define what are the parameters and their order. This will be used by the programmer when he calls the template in a specification file; it is important to notice that some parameters have a specific meaning inside the PSL such as the Aspect parameter.

The Execution variable is used when we have a pointcut that is passed as a parameter and we have to decide whether we execute a block that keeps the result (and probably does something with it) or if we only execute the pointcut without keeping the result; this is decided by the return type of our pointcut (if the return type is absent we assume it is void).

We use Template to say that it is a Pure Template file and then, inside that scope, we define several blocks of code. The most important block of code is the CORE block (this grammar is also case sensitive); this is the main block that gives the template its true meaning. The other blocks can be given any name as long as they start with upper case letters.

The only possible values for blocks are strings and blocks are the only type of variables that can be created inside the scope of a template.

3.3.2.2 Compositional Templates

The specific (informal) structure of these templates is:

```
Parameters = [PAR1, PAR2, ..., PAR3];

Composition {

        var1 = isomething';

        TemplateA<PAR1, var1, ..., PARx, varx>;
        (....)
        TemplateN<PARy, ..., PARn>;
}
```

The parameters are the same. The Composition keyword is used to say that this is a Compositional Template. These templates can only have two kinds of statement: variable assignments and template invocation. The variables can only be given strings. The templates can be invoked with parameters of the Parameters variable, with strings and with variables. The generation of output files is the same as it is with the specification files; the built-in templates are generated in the same way and the external templates generate always an output file.

3.3.2.3 Reserved keywords for Template tags

There are some tags that have a specific meaning inside of a template. These tags will always be replaced, even if they are not parameters of the template, and have the highest priority.
The reserved keywords for tags are:

- **<MAIN>**: This tag is replaced by the main method of an application; this has to be defined in the specification file.
- **<ASPECT>**: This represents the filename for the file that will be generated. It should be passed as a parameter.
- **<RT>**: The return type of the pointcut.
- **<RT_FAKE>**: Sometimes it is needed to create a fake object with the same type as the return type of the pointcut and this tag provides this functionality.
- **<METHOD>**: It is the name of the method of the pointcut.
- **<ARGS>**: It generates the corresponding args in AspectJ with the variable names.
- **<ARGS_IDENT>**: It puts together, separated by comas, the identifiers that are passed with the pointcut (or if the PSL generates automatically variable names, it has to put them here too).
- **<ARGS_LIST>**: It constructs a list with the variable types of the pointcut.
- **<ARGS_TYPES>**: List with the types of each variable of the pointcuts.
- **<,ARGS_LIST>**: Same as ARGS_LIST but adds a coma before the args_list.
- **<PRE>**: It is used by the composition mechanism. It should be put where we want to prevent other aspects from intercepting the call. The name comes from "pre-condition".

3.3.2.4 Parameters

The parameters that can be passed onto the templates can have any name as long as they are not keywords. The parameters are replaced after the keywords. The language does not allow, for now, passing multiple pointcuts onto the templates; you can have only one pointcut as a parameter.

The parameters are used in the body of the template by using them as tags; <PARAMETER_NAME>.

3.3.2.5 Execution block

The execution block needs 3 things:

- A name;
- The block to be used if return type of the pointcut is not void;
- The block to be used if the return type of the pointcut is, or is assumed to be, void.

Then, to use it in your file, you have to write the tag with the execution name, like <EXECUTION_NAME>, and you need to create the two blocks with the specific code for both situations. It is possible for the block of the "return" to use the execution of the "void" but the opposite is not possible.

A simple example:

```
Execution = Ex<Returns, Void>;

(...)

Block1 = 'if (true){ <Ex> }';
Returns = 'some <Void>';
Void = 'code';
```

3.3.2.6 Understanding template files

In this section we will show examples of template files and explain them.

```
Parameters = [ASPECT, PRE, T, P];

Template {
      CORE = '
      import concurlib.futures.*;

      public aspect <ASPECT> extends FutureProtocol {

            protected pointcut asyncMethodCall(Object servant) :
call(<RT> <T>.<METHOD>(..)) <PRE> && target(servant);

            protected pointcut useValuePoint(Object servant) :
call(<RT> *.waitForFuture(<RT>)) <PRE> && args(servant);

            protected Object getFakeObject() {
                  <RT> ax = <RT_FAKE>;
                  return(ax);
            };

      }';
}
```

- **Parameters = [ASPECT, PRE, T, P]** defines the order and name of the parameters that will serve as input to the template. *ASPECT, PRE, T* and *P* are all keywords of the language.
- **Template { ... }** is the template itself; it is used the keyword template to define this template file as a *pure template*.
- **CORE = ' ... '** is the main block of code of a template and it is the only one that is required.
- **public aspect <ASPECT> extends FutureProtocol** is the definition of this aspect with the name *ASPECT* that was passed as a parameter and that it extends another aspect named FutureProtocol.
- **protected pointcut asyncMethodCall(Object servant) : call(<RT> <T>.<METHOD>(..)) <PRE> && target(servant)** is defining a pointcut and it uses the tags *<RT>* (return type), *<T>* (Type /Class) and *<PRE>*. *T* is passed as a parameter and *RT* is generated from the parameter *P* (the return type of the pointcut). *PRE* is a "pre-condition" used by the composition mechanism.
- Inside the getFakeObject method there is **"<RT> ax = <RT_FAKE>"**. This is creating a variable named ax using the return type of the pointcut and it is assigning a fake value to the variable.

```
Parameters = [ASPECT, PRE, T, P];
Execution  = EX1<RT_BLOCK, EX_BLOCK>;

Template {
 CORE = ' ...
          public aspect <ASPECT> {

            ...

          public interface I<T> extends Remote {
                  public <RT> <METHOD>(<ARGS_LIST>) throws
                  RemoteException;
                  ...
          }

            ...

            Object around() : call (<T>.new(..)) && <MAIN> {
                  ...
            }

            ...

            <RT> around(<T> obj<,ARGS_LIST>) : call(<RT>
<T>.<METHOD>(<ARGS_TYPES>)) <ARGS> <PRE> && target(obj) {
                  ...
                  <EX1>
            }


 RT_BLOCK = '
      <RT> ax=null;
      try{
            ax = <EX_BLOCK>
      }catch (Exception e){e.printStackTrace();}
      return(ax);
 ';
}
 EX_BLOCK = '((I<T>) remotes.get(obj)).<METHOD>(<ARGS_IDENT>);';

}
```

- **Execution = EX1<RT_BLOCK, EX_BLOCK>** defines an execution block named **EX1**, a "return type" block named **RT_BLOCK** and an execution block called **EX_BLOCK**. RT_BLOCK and EX_BLOCK are not predefined names; they can have any name besides CORE.
- public *<RT>* **<METHOD>(<ARGS_LIST>)**, *METHOD* uses the method name of the pointcut and *ARGS_LIST* uses the list of parameters from the same pointcut.
- Object around() : call (<T>.new(..)) && **<MAIN>** is using *MAIN*, an ambient variable, and it is not required for this to be passed as a parameter.
- <RT> around(<T> obj<,ARGS_LIST>) : call(<RT> <T>.<METHOD>(**<ARGS_TYPES>**)) **<ARGS>** <PRE> && target(obj) is using *ARGS_TYPES*, because it only needs the types of the parameter used by the pointcut. *ARGS* is the name of those parameters.
- **<EX1>** says that the execution block will be placed here.

- **RT_BLOCK** defines the instructions to be executed inside that block; it is the block used by the execution block EX1 when the pointcut has a return type (different from void).
- **EX_BLOCK** = '((I<T>) Remotes.get(obj)).<METHOD>(<ARGS_IDENT>);' defines the method that will always be called inside the execution block, whether the pointcut has a return type or not.

```
Parameters = [ASPA, ASPB, PRE, T, PA, PB, SEP];

Composition {
      cod = '1234';
      fun =  iscatterFactory.scatter';

      Replicate<<ASPA>, <PRE>, <T>, 2, Broadcast<<PA>>,
      Scatter<<PB>, fun, Reduce<'ReduceFactory.scReduce'>>>;
      AsyncFut<<ASPB>, <PRE>, <T>, <PB>>;
      Separate<'Teste3', <PRE>, <T>, <PB>>;
      OutraComposicao<<T>, cod>;
}
```

- **Composition** defines this as a compositional template
- **AsyncFut<<ASPB>, <PRE>, <T>, <PB>>** is using four parameters. To use parameters you must write as *<NAMEOFPARAMETER>*.
- The rest of the syntax is similar to a specification file. It only defines strings and calls several templates with the given parameters. The *ASPECT* tag can be used as a parameter but it is meaningless because this type of templates does not generate a separate file for itself; it only generates files for the templates inside.

3.3.2.7 Possible improvements to the design

There are some things that are known to be missing in the present specification of the PSL that can and should be added to it because of its importance.
Some important things are:

- The possibility of having multiple pointcuts as parameters of a template
- Several execution blocks in the Pure Template files
- When passing pointcuts to templates it should not be needed to pass the names of the variables along with the types.
  - The PSL should have mechanisms for the automatic creation of variables and there should be a way for the templates to use those variables if they needed them.
- It should also be possible for pointcuts to be defined using wild cards in the same way that they are in AspectJ
  - This was not done because the implementation of this leads to many problems and it is very time consuming. For this to be possible it is needed to create a pattern matcher, analyze the classes that use the methods that are compatible with the pattern and the, because the implementation avoids the use of proceeds, it needs to generate lots of different blocks of execution with the different options of the pointcut. Now that the notion of blocks and templates have been created and used, this could be easier to do.
- There should be special constructs in the PSL when using distribution so that it would be possible to recognize and handle communication problems and implement fault-tolerance.
- There should be a theoretical model and a formal specification of the PSL, especially for the composition mechanism and substitution algorithm; all is done in a very pragmatic way so there is no proof that it works well for every case.
- There are templates that depend on the type and number of the parameters of the pointcut; this makes it very hard (or even impossible) to do a generic template without a proper mechanism to handle this. A solution was thought for this problem but it was not implemented due to lack of time. The possible solution for this was for the PSL to generate tags for each of the parameters (like <PCUT_PAR[0]>, <PCUT_PAR[1]>, ... <PCUT_PAR[N]>). It would also be helpful if the PSL generated an array of objects with the parameters inside. The scatter template does something similar but this is an internal template with a hard-coded generation method.

3.4 Implementation

3.4.1 Used tools

The tools that were used in the development of the grammar and the parsers were Eclipse 3.2, ANTLREclipse plug-in, AspectJ plug-in for Eclipse, SVN repositories and Java 5.0 SDK.

The grammars have been implemented in ANTLR 2 and the parsers are implemented in Java. ANTLR was chosen because it had already been used in the project for other tasks and the parsers are done in Java because we needed the parsers to be platform-independent. AspectJ is used in the definition of the templates; it is used to show how AOP can be used to produce parallel and distributed applications even if AspectJ by itself is very sloppy in handling most cases; this was surpassed by building a tool on top of AspectJ that is aware of its shortcomings and flaws.

3.4.2 Installation and Running

To get the source code and/or the binaries you should get them from here:

http://gec.di.uminho.pt/ppc-vm

You must have ANTLR 2.6 and Java 5.0 SDK installed to be able to compile the source code files.

To run the PSL you just need to run the Main class file: java Main
To compile a specification file you must have the MakeGenerics file in the same directory as the PSL compiler and all the templates in a folder called Templates (be sure that the T is upper case).

3.4.3 Achievements

The tool is stable and fully working and can generate error-free AspectJ code given that the templates used are themselves error-free. It implements all the features that were described along this report.

3.4.4 Initial problems

The PSL encountered several problems at the beginning of its development. We had very little experience with the use of ANTLR, which proved to be very problematic, but also with the design and implementation of the PSL.

It was easy to see, even in the beginning, that the templates were going to be essential and that they needed to be separated from the PSL. As there were no previous study on the patterns of the templates (or even templates), it was impossible at first to do or even think about generic templates, so there was a need to hardcode several different generation algorithms for different templates.

When we were implementing those algorithms we felt that we needed to separate some parts so it would be easier to compose them (and re-use them several times, if needed) and join them all in one file; that lead to the code blocks.

After implementing several templates some tags became evident and we decided to turn them into keywords. Also, when we hard-coded the different templates, we encountered problems doing the replacements and we used that knowledge for the generic templates.

Even after all these experiments, we had not consider some things until we had the PSL being used in real programs; one problem that we had was when we were using the broadcast and we used a pointcut for a void method and it did not work because the PSL was always saving the results to an hashmap; that lead to the development of optional code blocks (or execution blocks).

With the knowledge that we gained implementing the first version of the PSL, it became possible to have a stable processor for generic templates and a stable processor for the specification file processor. This work method shows that the language was done in a very pragmatic way, always gaining experience and knowledge by testing and implementing features and by using it in a real setting.

3.4.5 Brief overview and explanation of the classes

Some of the most important classes are the Command classes. These classes have the name **[TemplateName]Command** or **Command[TemplateName]** name and they extend the Command class. The **Command** class has some implemented methods but the most important feature is that it has the generateCommand as an abstract method that all the commands should implement. This forces the classes that inherit from Command to know something about the templates that they represent and forces them to implement the generation algorithm.

ExecBloc is a container class that is used to store the information of an execution block. It only has a small set of get methods and a constructor.

Execucoes is the class that creates the pre-conditions and that is responsible for the generation of the files of the built-in templates and it is also responsible for generating the file with the precedences.

Expressoes has several useful methods for strings and regular expressions.

The **Main** class is the class that compiles the specification file.

Template is the class that is used to represent a generic template and it has a built-in algorithm for generating files based on generic templates.

ComposionTemplate extends the template class to add composition.

TPLProcessor is the class that is used as a processor for the template files.

TextFile is the class that is used for loading and saving text files.

Metodos saves the information about the methods that were declared inside the specification file.

Variaveis is the class that holds all the info that the PSL collects from the specification files.

VariaveisTemplates is the class responsible for saving all the info that comes from generic templates.

GPT.g and **TemplateProcessor.g** are the ANTLR grammars that are used in this application.

3.4.6 What can be improved?

The implementation of the PSL, for this current specification, is still lacking some features.
This is a list of features that could be added in the implementation of the PSL:

- **Adding testing methods**
  - All the tests that were done when creating the PSL were done manually and most of them were lost and could not be used for documentation purposes. I would recommend the use of some best practices in the future, like:
    - Unit Testing using JUnit
      - The tests would serve as documentation of problems but also as regression tests when something new was added to the language.
    - Grammar Testing using GUnit
      - The development of a grammar leads to some problems that would be easily found if there was a tool like GUnit running. It also makes sure that by adding productions to the PSL that the grammar does not generate errors where there were not any (regression testing).
    - The use of Programming by Contract using JML
      - By asserting code and giving specifications to what should be done, it makes much easier to find the bugs and prevents the code from becoming tangled with the print statements that are usually used for debugging purposes.
  - After this practice is adopted I would recommend deleting all the print statements from the base code because they are only there for debug purposes and with this approach they become obsolete.
- **Re-implementation of the PSL in ANTLR 3**
  - The grammar was done in ANTLR 2 because it was what was available when we began working on it. ANTLR 3 provides several improvements to ANTLR 2 that could be used to turn the PSL more elegant and easy to understand and maintain.
- **Improving error messages.**
  - As it stands, the PSL does not provide a meaningful way to show errors. It only depends on the ability of the parser to recognize or not a given statement. There should be more meaningful error messages that truly showed what and where the compilation failed. There is also a problem with the line counting and it often reports the wrong line when there is an error.
- **Validation of values**
  - The PSL should check the strings that are used to see if they are valid classes, or valid functions or if it has valid AspectJ code inside of them, depending on which case applies. This would add great value to the PSL and would surely improve the static checking of it.

- **Enforcing coding conventions**
  - It would greatly help the understanding of the code if it was developed following always the same coding conventions.
- **Tools for documentation**
  - The code was not commented properly and it is lacking proper documentation in the API has it is in the PSL.
- **Graph of dependencies**
  - It would also be helpful to have a graph showing how all the templates of the specification file are connected.
    - This could be skipped if there was a visual programming tool for the PSL instead of a text tool.
- **Tool for visual programming**
  - This tool would allow connecting several templates only by visually creating the objects and drawing the connections between them. It would be great if there were integration with an IDE like Eclipse and if it would allow to drag-and-drop functions and classes right into the tool.

Appendix

A – Grammars

A.1 - Specification Files Parser

```
start :
      VAR LCURL
      ( expressoes )*
      RCURL
      ;

expressoes :
      atrib
      | method_decl
      | aspect_creation
      | variable_insertion
      | stat
      ;

variable_insertion :
      VARINS MENOR STRING VIRG VAR VIRG STRING MAIOR FIM
      ;

aspect_creation :
      ASP VAR ATRIB MENOR values
      ( VIRG values )?
      MAIOR FIM
      ;

method_decl :
      METH MENOR VAR VIRG STRING MAIOR FIM
      ;

atrib :
      atrib_var
      | atrib_var_amb
      ;

atrib_var :
      VAR ATRIB STRING FIM
      ;

atrib_var_amb :
      PACKAGE ATRIB STRING FIM
      | MAIN ATRIB STRING FIM
      ;

stat :
      comando FIM
      ;

comando :
```

```
      REPLICATE MENOR VAR VIRG values VIRG
      ( comando | values )
      VIRG INT
      ( lista_replicate )?
      MAIOR
      | IDENT MENOR lista_param MAIOR
      ;

lista_param :
      values
      ( VIRG lista_param )?
      ;

lista_replicate :
      VIRG opcoes_lista
      ( lista_replicate )?
      ;

opcoes_lista :
      DELEGATE MENOR values MAIOR
      | BROADCAST MENOR values
      ( VIRG REDUCE MENOR values MAIOR )?
      MAIOR
      | SCATTER MENOR values VIRG values
      ( VIRG REDUCE MENOR values MAIOR )?
      MAIOR
      |
      ( AFTER | BEFORE )
      MENOR values VIRG values MAIOR
      ;

values :
      STRING
      | VAR
      | MAIN
      ;

mini :
      ( miniC FIM )*
      ;

miniC :
      REPLICATE MENOR values VIRG values VIRG
      ( miniC | values )
      VIRG INT
      ( lista_replicate_mc )?
      MAIOR
      | IDENT MENOR lista_param MAIOR
      ;

lista_param_mc :
      MENOR values MAIOR
      ( VIRG lista_param_mc )?
      ;

lista_replicate_mc :
      VIRG opcoes_lista_mc
      ( lista_replicate_mc )?
      ;
```

```
opcoes_lista_mc :
      ( DELEGATE MENOR values MAIOR
      | BROADCAST MENOR values ( VIRG REDUCE MENOR values MAIOR )?
MAIOR
      | SCATTER MENOR values VIRG values ( VIRG REDUCE MENOR values
MAIOR )? MAIOR
      | ( AFTER | BEFORE ) MENOR values VIRG values MAIOR )
      ;
```

A.2 – Specification Files Tree Parser

```
start :
      START
      ( expr )*
      ;

expr :
      ATRIB_VAR VAR STRING
      | ATRIB_AMB
      ( PACKAGE | MAIN )
      STRING
      | meth_decl
      | aspect
      | stat
      ;

aspect :
      ASPECT VAR STRING
      ( STRING )?
      ;

stat :
      STAT comm
      ;

comm :
      ( COMM REPLICATE VAR valor ( valor | comm ) INT lr ) =>
      COMM REPLICATE VAR valor
      ( ( valor ) | ( comm ) )
      INT lr
      |
      ( COMM IDENT lista_param ) =>
      COMM IDENT lista_param
      ;

mini :
      MINI
      ( miniC )*
      ;

miniC :
      COMM
      ( REPLICATE valorn valorn ( ( valorn ) | ( miniC ) ) INT lr |
IDENT lista_param )
      ;

lista_param :
```

```
      LP valor
      ( lista_param )?
      ;

meth_decl :
      METHOD VAR valor
      ;

lr :
      LR op_lr lr
      |
      ( )
      ;

op_lr :
      ( DR BROADCAST valor ( valor )? ) =>
      DR BROADCAST valor
      ( valor )?

      | ( DR DELEGATE valor ) =>
      DR DELEGATE valor
      |
      ( DR SCATTER valor valor ( valor )? ) =>
      DR SCATTER valor valor
      ( valor )?

      | ( DR AFTER valor valor ) =>
      DR AFTER valor valor
      |
      ( DR BEFORE valor valor ) =>
      DR BEFORE valor valor
      ;

valor :
      STRING
      | VAR
      | MAIN
      ;

valorn :
      STRING
      | VAR
      | MAIN
      | IDENT
      ;
```

A.3 – Specification File Lexer

```
MENOR :
      '<'
      ;

MAIOR :
      '>'
      ;

VAR :
      ( 'a'..'z' )
      ( 'a'..'z' | 'A'..'Z' | '0'..'9' | '_' )*
      ;

IDENT :
      ( 'A'..'Z' )
      ( 'a'..'z' | 'A'..'Z' | '0'..'9' | '_' )*
      ;

INT :
      ( '0'..'9' )+
      ;

VIRG :
      ','
      ;

DOLLAR :
      '$'
      ;

STRING :
      ( '\'' ( . )* '\'' )
      ;

LBRACK :
      '['
      ;

RBRACK :
      ']'
      ;

LCURL :
      '{'
      ;

RCURL :
      '}'
      ;

PONTO :
      '.'
      ;

ATRIB :
      '='
      ;
```

```
FIM :
      ';'
      ;

WS :
      ( ' ' | '\r' '\n' | '\n' | '\t' )
      ;

LINE_COMMENT :
      '#'
      ( ~( '\n' | '\r' ) )*
      ( '\n' | '\r' ( '\n' )? )?
      ;
```

A.4 – Template Files Parser

```
start :
      PARAMLIST IGUAL LBRACK lista_param RBRACK FIM executions
            ;

executions :
      ( EXECUTION IGUAL blkExecucao FIM )?
      blocks
      ;

blkExecucao :
      IDENT MENOR IDENT VIRG IDENT MAIOR
      ;

lista_param :
      tag resto_lista
      ;

resto_lista :

      | VIRG tag resto_lista
      ;

tag :
      IDENT
      ;

block :
      IDENT IGUAL STRING FIM
      ;

blocks :
      TEMPLATE LCURL lista_blocks RCURL
      | COMPOSITION LCURL
      ( expressoes )+
      RCURL
      ;

expressoes :
      ( atrib | call )
      FIM
      ;

atrib :
      VAR IGUAL STRING
      ;

call :
      ( REPLICATE MENOR values VIRG values VIRG ( call | values ) VIRG
values ( opcoes_replicate )? MAIOR | SEPARATE MENOR values VIRG values
VIRG values VIRG values MAIOR | IDENT MENOR param_call MAIOR )
      ;

param_call :
      values
      ( VIRG param_call )?
      ;
```

```
values :
      STRING
      | VAR
      | IDENT
      | INT
      | MENOR IDENT MAIOR
      ;

opcoes_replicate :
      VIRG opcao
      ( opcoes_replicate )?
      ;

opcao :
      ( DELEGATE MENOR values MAIOR | BROADCAST MENOR values ( VIRG
REDUCE MENOR values MAIOR )? MAIOR | SCATTER MENOR values VIRG values
( VIRG REDUCE MENOR values MAIOR )? MAIOR | ( AFTER | BEFORE ) MENOR
values VIRG values MAIOR )
      ;

lista_blocks :
      block
      ( lista_blocks )?
      ;

resto_lblock :
      block
      ( resto_lblock )?
      ;
```

A.5 – Template Files Tree Parser

```
start :
      START
      ( lista_param )
      ( executions )
      ;

lista_param :
      LP tag resto_lista
      ;

executions :
      EXEC
      ( blkExecucao )?
      blocks
      ;

tag :
      IDENT
      ;

resto_lista :
      RL tag resto_lista
      |
      ;

blkExecucao :
      BLKEXEC IDENT IDENT IDENT
      ;

blocks :
      BLKS
      ( TEMPLATE lista_blocks | COMPOSITION expressoes )
      ;

lista_blocks :
      LB block
      ( lista_blocks )?
      ;

expressoes :
      ( exps )+
      ;

exps :
      EXP
      ( atrib | call )
      ;

atrib :
      ATRIB VAR valor
      ;

call :
      CALL
```

```
      ( REPLICATE valor valor ( ( valor ) | ( call ) ) valor
opcoes_replicate | SEPARATE valor valor valor valor | valor param_call
)
      ;

block :
      BLK IDENT STRING
      ;

valor :
      STRING
      | VAR
      | IDENT
      | INT
      | MENOR IDENT MAIOR
      ;

opcoes_replicate :
      OR opcao opcoes_replicate
      |
      ;

param_call :
      CP valor
      ( param_call )?
      ;

opcao :
      OP
      ( BROADCAST valor ( valor )? | DELEGATE valor | SCATTER valor
valor ( valor )? | AFTER valor valor | BEFORE valor valor )
      ;
```

A.6 – Tempate Files Lexer

```
MENOR :
      '<'
      ;

MAIOR :
      '>'
      ;

SLASH :
      '/'
      ;

BACKSLASH :
      '\\'
      ;

ASTERISK :
      '*'
      ;

CARD :
      '#'
      ;

IGUAL :
      '='
      ;

LBRACK :
      '['
      ;

RBRACK :
      ']'
      ;

LCURL :
      '{'
      ;

RCURL :
      '}'
      ;

FIM :
      ';'
      ;

VIRG :
      ','
      ;

STRING :
      ( '\'' ( . )* '\'' )
      ;

WS :
```

```
        ( ' ' | '\r' '\n' | '\n' | '\t' )
        ;

LINE_COMMENT :
        CARD
        ( ~( '\n' | '\r' ) )*
        ( '\n' | '\r' ( '\n' )? )?
        ;

INT :
        ( '0'..'9' )+
        ;

VAR :
        ( 'a'..'z' )
        ( 'a'..'z' | 'A'..'Z' | '0'..'9' | '_' )*
        ;

IDENT :
        ( 'A'..'Z' )
        ( 'a'..'z' | 'A'..'Z' | '0'..'9' | '_' )*
        ;
```

B – Syntax Handbook

B.1 – Specification file

The types of statements we can do with the PSL are:

- **Assignments**
  - You can assign values to any variable
    - There is not a notion of constant in the PSL
  - There are two types of variables
    - Environment variables. These have specific meaning inside the PSL and should not be used for other purposes
    - User-defined variables.
  - It uses the operator =
    - var1 = 'hello world'
- **Declaration of variables using primitive types only**
  - There are 3 primitive types: Aspects, strings and Methods.
    - Strings can be multi-line and they begin and end with '
      - 'Hello
        World'
    - Aspects are declared with their value but are considered to be constant
      - The first value is the name that will be used for the generated file and the second value is the package
        - Aspect asp1 = <'AspectFile', 'ppcvm.PSL'>
        - Aspect asp2 = <'Aspect2'>
    - Methods are declared with two parameters: the aspect where they are going to be inserted and the whole method.
      - Method<asp1, 'void print(){
        System.out.println("hello");
        }'>
  - Every type uses strings but in different ways.
  - Generic variables are declared without any type and only aspects and methods need to be declared with the type name.
  - All identifiers are forced to begin with lower case
    - Unless they are a specific environment variable such as MAIN
  - The PSL is case sensitive

- **Execution statements**
  - Always begin with upper case letters
  - The identifier name has to be identical to an existing template
  - An execution statement is written as
    - TemplateName< parameter1, ..., parameterN>
- **Comments**
  - There are only single line comments
    - # This shall be ignored by the compiler
- Every statement is terminated by the ;