Computer Science Department
University of Valladolid
Valladolid - Spain

SPC-XML(v0.4): An intermediate programming language representation for nested-parallel environments

Arturo González-Escribano[1], Arjan J.C. van Gemund[2], Valentín Cardeñoso-Payo[1], and Raúl Portales Fernández[1]

[1] Dept. de Informática, Universidad de Valladolid.
E.T.I.T. Campus Miguel Delibes, 47011 - Valladolid, Spain
Phone: +34 983 423270, eMail:`arturo@infor.uva.es`

[2] Embedded Software Lab, Software Technology Department,
Faculty of Electrical Engineering, Mathematics and Computer Science, P.O.Box 5031,
NL-2600 GA Delft, The Netherlands
Phone: +31 15 2786411, eMail:`a.j.c.vangemund@ewi.tudelft.nl`

Feb 2005

Abstract. This document presents a parallel programming language based on XML, which can be useful as an intermediate representation for nested-parallel programs. It is a coordination language which clearly makes a separation between the parallelism specification and the sequential code pieces. The design criteria of this language are oriented to make it specially useful on nested-parallel programming environments where scheduling and other compiling techniques need to obtain synchronization structural information from the program specification.
This specification supersedes preliminary versions of SPC-XML (from v0.1 to v0.3)

Technical Report No. IT-DI-2005-0001

1 Introduction

SPC-XML is an XML application which defines a parallel programming specification language. A first draft and prototype programming frameworks which implements the preliminary versions of this language were introduced on [1, 2, 5].

This document presents a transitional proposal of the language. SPC-XML v0.4 is only a draft designed to help in detecting problems and solutions for the development of a full compiler-framework supporting SPC-XML. In fact it includes some features which may be problematic for implementation. A new version of SPC-XML will be presented when the possibilities and requirements of the compiler-framework becomes more defined.

This report is structured as follows: Section 2 introduces the design principles and structure of the language, along with a discussion of data representation, memory model and some other semantic details. Section 3 includes an example program. In section 4 we present our conclusions. Appendix A contains a fully detailed list of SPC-XML v0.4 tags. Appendix B contains the full DTD (Data Type Definition) which is the XML formal description of the language tags. Appendix C describes some changes from the previous version.

2 SPC-XML: A tag-based coordination language

In this section we introduce the structure and design principles of the a new intermediate highly-structured coordination language based on XML. It is named SPC-XML, due to the SPC nested-parallel model [4] (Series-Parallel and Contention model). This full parallel synchronization language is designed to support any feature to be found in a nested-parallel environment such as recursion, critical sections, distributed or shared-memory models, and manual data-layout specifications. The language is highly verbose and it is not designed to be written directly by a programmer, but as a convenient intermediate representation of a nested-parallel program. Front-ends to translate legacy code written in any nested-parallel language to this representation would be a straightforward development effort. Nevertheless, standard XML tools may be used to edit, visualize or check consistency of SPC-XML representations. Although its main functionalities and semantics are clearly defined, SPC-XML is still syntactically evolving to find a more mature level. The design principles of SPC-XML are:

1. SPC-XML model implements the same semantics as the SPC model [4]: SPC model is based on a processes algebra. Processes are composed to form a program with only two possible operators – sequential and parallel –, which may be freely nested.
2. It uses XML tags to represent directly the parallelism structure. Nested-Parallelism is a hierarchical structured form of expressing parallelism. Thus, a representation using an XML structured document is natural.
3. It is a coordination language [3]. Tags will represent only the synchronization structure. Any classical sequential language may be used to code the

sequential sections. This separation will simplify the recognition of the parallel structure of a program.

4. The program must specify explicit input/output interfaces for sequential sections and logical processes. The behavior of a data item in an interface (input and/or output) will be compulsory specified. This will help the compiler to compute data-flow and automatically derive communications for a given task decomposition and mapping.
5. Tag and attribute names are fully readable, to help humans recognize the main program structures easily.
6. SPC-XML supports the synchronization mechanisms of any other nested-parallel language, to be useful as an intermediate representation for legacy programs.

2.1 Document structure

An SPC-XML document contains a HEADER tag and a UNIT-BODY tag. The first one is used mainly for documentation and to specify low-level sequential code blocks to be included at the beginning of the target program. The body of the document contains a collection of elemental units (functions and processes). If a process called main is found inside a document, this document represents a program which begins executing that main process. If the document does not include a main process, it is a library which may not be compiled alone. All elemental units may have a DESCRIPTION tag containing extra documentation tags for automatic documenting tools. Programming comments are included with the usual XML tags <!-- -->.

2.2 Logical processes and functions

SPC-XML has a PROCESS tag to define the content of a logical process. Each process has a well-defined interface, whose formal parameters are defined with the IN, OUT, INOUT tags, declaring the input/output intent of the data item in the interface. Processes may contain also a LOCAL tag to declare variables, with process visibility and scope. The BODY tag of a process contains the tags which define its behavior.

The serial composition is implied in the tags declaration order as in many common procedural languages. For programmer convenience, other common control-flow statements are supported by the IF/THEN/ELSE, WHILE, REPEAT, and LOOP tags. The parallel composition operator is implemented using the PARALLEL tag. This tag may only contain one or more PARBLOCK tags, whose contents are composed in parallel. The PARALLEL tag, have an optional attribute named p="number", to specify a fixed number of branches to spawn. The PARBLOCK tags may also have a index="number" attribute, to associate the tag content with a logical identifier inside the local subgroup of branches. A default branch (using index="*") may be used to fill-up with its content all the non-associated branches in a parallel region. The closing parallel tag implies a synchronization of all the branches before proceed. At this logical synchronization point, modifications done to the same variable in different branches are made consistent. By default, the modifications done by only one branch will persist. Future specifications should include optional attributes or tags to be used in the PARALLEL tag to implement typical reduction operations for given data elements. As in any nested-parallel model, other PARALLEL tags may appear inside a branch, or inside another process called inside a branch. All control-flow and parallel structure tags may be nested freely.

Sequential codes are enclosed inside a FUNCTION tag, to distinguish them from the logical processes. However, the input/output interface is defined exactly in the same way. For symmetry, processes and functions are invocated with the same CALL tags, containing PARAMETER tags to specify the formal to real parameter substitutions.

The language includes some more features to help in the mapping trajectory. For instance, the programmer, or a profiling tool, may provide sequential functions with a performance estimation, using an optional workload="number" attribute, as a hint to help the scheduling modules.

2.3 Data representation

As a coordination model, SPC-XML tags do not imply data manipulation. The only purpose to define data containers is to describe data-flow between logical processes and through sequential codes. Thus, SPC-XML variables are generic multi-dimensional arrays of a given data type defined in the sequential language used. The actual dimensions of a variable are readable attributes. They are defined by VAR tags, always inside the LOCAL tag of a process. There are no expression or data manipulation tags. The only tags which imply code execution are CALL tags referring to function names (sequential tasks). However, we also include a COPY tag, whose purpose is to let the programmer specify data-flows in a way that such information may be exploited by an SPC-XML compiler. This tags will be translated to memory-to-memory copies, or directly to communications. Special variables called *overlaps* are supported to define logical data partitions over other SPC-XML array variables. Thus, simple data-layouts may be devised. An overlap is defined and used as any other SPC-XML variable. However, before using it, it must be associated to another variable using an VAR-OVERLAP tag. Each overlap element becomes an alias of a subarray of the associated SPC-XML variable. Subarray specifications may be expressed with a *colon-notation* similar to Fortran90 and simplified by the use of some macro-definitions. Moreover, the language provide with some special terms which represent typical block or stride layouts.

2.4 Memory model

The SPC-XML semantics are based on a distributed-memory model. Each process has access only to their local variables and parameters. The parameter are always received as values. Thus, a process always works on local copies of the data.

Communication is hidden to the programmer on the parameter substitution when an invocation to a process is executed in another processor/thread. However, when the programmer knows that, in a shared-memory architecture, it is safe to work on the original memory positions instead of a local copy, she may specify the `shared="true"` attribute for that variable. This attribute value is used by the compiler as a hint for efficient implementation when using a shared-memory back-end. Nevertheless, the programmer may not rely on this feature to communicate processes/threads (e.g. one thread writing on a variable and another reading the values directly from the memory positions). First, the compiler does not consider synchronization issues for these variables. And much more important; the underlying architecture or back-end may not support shared-memory, implementing local copies anyway. Allowing such programming practices would destroy the portability premises of the framework.

2.5 Static vs. dynamic programming

Nested-Parallel program specifications may have three different flavors, presenting implementation problems which should be tackled with different approaches. A flexible compiler should detect them and use different compilation techniques for each one. We describe the three types, and how SPC-XML parsing may automatically detect them.

Many times, the data-partition needed to exploit parallelism for a given computation is specified statically as a function of the number of processors (e.g. the simple pipeline example in section 3). These programs resemble the typical coarse-grain efficient specifications found in lower-level specifications, such as those done for message-passing tools. These specifications do not normally need more than one level of parallelism, but if needed, the scheduling and mapping may be computed at compile-time.

On the other hand, divide & conquer and other similar algorithmic design techniques decompose the data recursively. Examples of these applications include recursive varieties of FFT, Strassen matrix multiplication or some sorting networks. These fine-grain approximations have a higher abstraction level, and they are more flexible. The compiler may decide the exact grain to use, stopping the recursive parallel decomposition when it becomes appropriate, to adapt the computation to the machine or environment. This complex decisions may be taken at compile-time.

Finally, some solutions are based on dynamic parallel structures, which are created and destroyed as the data-process require them. Thus, they are completely data-dependent. In these cases, the scheduling and load balance may be done only at run-time. Examples of these programs may be some hierarchical n-body solutions or tree/graph-searching.

These different types of programs may be easily detected in a coordination language with the design principles assumed in SPC-XML: (a) Static problems do not have control-flow structures (except loops with a fixed number of iterations) which affect the coordination structure (with parallel tags inside); (b) recursive, but not fully dynamic decomposition, present control-flow statements whose conditions are dependent on the data-sizes, but not on their values; (c) dynamic solutions have control-flow statements, dependent on the data values, which affect the coordination structure. Thus, SPC-XML parsing may easily detect these different types of programs, enabling different compilation-paths or mapping modules for each type.

2.6 Data partitions (layouts)

We distinguish three main types of data partitions depending on the compilation/run-time level on which layout decisions should be taken:

Completely static :
In completely static data partitions the number of parts in which data are splited is fixed, and the sizes of each part is also known at compile time.
This situation could easily arise when using SPC-XML as an intermediate representation of another language. The front-end which generates the SPC-XML code from the source code may compute even the part sizes.

Recursive dynamic : Typical divide&conquer algorithms divide their data accordingly to an heuristic. But the number of pieces in each level is fixed (e.g. quicksort algorithm splits the array in two parts at each level). What is computed dynamically is the size of each part. However, the number of spawned subprocesses cannot be known at compile time. In this sense, the synchronization structure is dynamic, and some run-time scheduling policy will be needed.

Structural static :
A similar situation appears when the the number of parts is fixed in the code (or by the first compiler stages), but the number of data pieces on each part is data-dependent or computed on scheduling or run-time, even if the partition is not recursive. In this case, the synchronization structure of the program is static, even if the data sizes are not, and the scheduling may be computed before run-time.

Completely dynamic : When non-recursive data partitions are used, is typical that the one-level partition is done in as many parts as processors are available. If the decision of the number of processors available is delayed until scheduling or run-time, the number of parts (and also their sizes) cannot be fixed in the source code.
This situation also appears when several levels of parallelism are used, but the decision of how many processors are assigned to each group is taken by the scheduler or mapping module.

A flexible compiler should be able to deal with each kind of data partition or layout at the appropriate level. As many decisions as possible should be taken before run-time to enable more sophisticated mapping techniques. An efficient run-time data partition and scheduling policy should be provided for last chance.

2.7 Internals of an overlap variable

An overlap variable should be implemented in a way that each part is a complete reference to the original variable pieces defined during the overlap declaration. The semantics of the overlapping model imposes that an overlap is a reference to the original variable. Thus, accesses through different overlaps of the same variable always read and write on the memory locations of the original variable.

Overlaps A'' of a previous overlap part $A'[i]$ are allowed. The new overlap A'' will reference to the memory positions of the root variable A, although the sizes and positions are computed according to the layout specification on the intermediate overlap A'.

Overlaps are mainly used to create data-layouts, in order to split data along processes. Thus, the possibly sparse memory allocations referenced by an overlap part should be efficiently marshalled to be communicated when needed.

Back-ends using a C-like language as native/target language should create the appropriate pointer structures to implement overlaps. When overlap parts are formed by consecutive data pieces, one pointer and one storage size would suffice to represent the part position (e.g. row block partitions). However, when *stride* partitions are created, or multiple-dimension variables are used, the data pieces of an overlap part may not be consecutive. Then, each part should be represented by a list of structures containing pointer arrays and sizes on each dimension.

On the other hand, Fortran90 like languages, which allow named multidimensional array pointers, provide direct support for overlaps. Fortran77 back-ends are not possible as the target language does not provide pointers, and it is not able to represent overlaps.

To avoid all these problems, the back-end designers should consider to use any other portable mechanism. For example, the MPI library provides procedures to create and destroy new data-types which could be used to define overlaps.

3 Examples

3.1 Simple pipeline (static data-layout)

This example is a parallel implementation of the following inner loop:

```
double v[800000];

for(iterations=1; iterations<10000; iterations++) {
    for(i=1; i<800000; i++) {
        v[i] = (v[i] + v[i-1]) / 2;
        }
    }
```

It is an example with an 800000 elements vector statically splited among P parallel tasks. The example uses the C `dran48` function to initialize the data with a given seed found in a file named `pipe.seed`.

```
<SPC-XML>
<HEAD>
    <TITLE>Pipe</TITLE>
    <DESCRIPTION>...</DESCRIPTION>
<LANG name="C" />
</HEAD>

<UNIT-BODY>

<!-- Specific C includes -->
<CODE>
#include<stdio.h>
#include<stdlib.h>
#include"myTypes.h"
</CODE>

<!-- Ancillary functions -->
<FUNCTION name="init">
    <DESCRIPTION>Initialize </DESCRIPTION>
    <PARAMETERS>
        <INOUT name="vector" basetype="double" dim="size">
    </PARAMETERS>
    <CODE>
        int i;
        FILE *fseed;
        long int seed;

        /* 1. READ THE SEED */
        if (NULL==(fseed = fopen("pipe.seed","r"))) {
                perror("Opening the file pipe.seed");
                exit(-1);
                }
        if (1!=fscanf(fseed,"%ld",&seed)) {
                perror("Reading the file pipe.seed");
                exit(-1);
                }
        fclose(fseed);

        /* 2. INIT RANDOM STREAM */
        srand48(seed);

        /* 3. RANDOM VALUES */
        for (i=0; i<(*size); i++) {
                vector[i] = drand48();
                }
    </CODE>
</FUNCTION>

<!-- Main computation -->
<FUNCTION name="compute">
    <DESCRIPTION>Simple forward loop</DESCRIPTION>
    <PARAMETERS>
        <IN name="firstStage" basetype="bool" dim="1">
        <IN name="border" basetype="double" dim="1">
        <INOUT name="data" basetype="double" dim="size">
    </PARAMETERS>
    <CODE>
        int i;

        /* 1. COMPUTE FIRST ELEMENT EXCEPT FOR FIRST STAGE */
        if (! firstStage ) {
                data[0] = (data[0] + border[0]) / 2;
                }

        /* 2. COMPUTE THE REST OF ELEMENTS*/
        for (i=1; i<(*size); i++) {
                data[i] = (data[i] + data[i-1]) / 2;
                }
    </CODE>
</FUNCTION>

<!-- Synchronization structure -->
<PROCESS name="main">
    <DESCRIPTION>Main: Data layout for 8 processors</DESCRIPTION>
    <PARAMETERS>
    </PARAMETERS>
    <LOCAL>
        <VAR name="data" basetype="double" dim="800000">

        <!-- Split data and overlap to border elements -->
        <VAR-OVERLAP srcVar="data" name="pieces[#P]" layout="[#blocks]" />
        <VAR-OVERLAP srcVar="data" name="borders[#P]" layout="pieces[#i][0]" />
        <!-- null border --> <VAR name="foo" basetype="double" dim="1">

    </LOCAL>

    <BODY>
        <!-- Init data -->
        <CALL name="init">
            <PARAM value="data" />
        </CALL>

        <!-- Iterations -->
        <LOOP n="10000">

            <!-- Parallel computations -->
            <PAR n="#P">

                <!-- First stage of the pipeline has no input border -->
                <PARBLOCK index="0">
                    <CALL name="compute">
                        <PARAM value="true" target="firstStage" />
                        <PARAM value="foo" target="border" />
                        <PARAM value="pieces[0]" target="data" />
                    </CALL>
                </PARBLOCK>

                <!-- Rest of the pipe stages uses previous stage border -->
                <PARBLOCK index="*">
                    <CALL name="compute">
                        <PARAM value="false" target="firstStage" />
                        <PARAM value="borders[#i]" target="border" />
                        <PARAM value="pieces[#i]" target="data" />
                    </CALL>
                </PARBLOCK>
            </PAR>
        </LOOP>
    </BODY>
</PROCESS>

</UNIT-BODY>
</SPC-XML>
```

4 Conclusion

This report introduces the SPC-XML v0.4 intermediate representation for nested-parallel programming languages. The design principles of this representation allow the compiler to exploit information about the synchronization structure of an application. Different mapping or scheduling techniques could be automatically selected as a function of the structural details of the application Moreover, information about data-flow between tasks may be directly obtained from the SPC-XML specifications. Implicit communication structures are also exposed and could be easily optimized, to generate low-level codes adapted to a specific target-machine.

A Summary of tags

A.1 General structure

<SPC-XML> </SPC-XML>

Opening and closing tags for any SPC-XML document. It contains one HEAD and one UNIT-BODY tags.

<!- - - ->

Comments are introduced using these sgml's standard tags for documentation.

<DESCRIPTION> </DESCRIPTION>

Tags containing data to be used by an automatic documentation program. This tags may be used inside HEAD, PROCESS and FUNCTION tags.
Proposed documentation tags: AUTHOR, DATE VERSION, ...

<HEAD> </HEAD>

Opening and closing tags for the head of an SPC-XML document. Head contains the program title, an indication of the sequential language used to implement sequential parts, and optional documentation description.

<TITLE> </TITLE>

This tag contains a title string for the program or library.

<LANG name="" />

This tag is empty. The required attribute `name` contains the name and version (if applicable) of the sequential language used to write the sequential code pieces of the program. It should be used by the compiler to select the appropriate back-end language module or plug-in.
Some standard names should be accepted by any compiler of SPC-XML even if the appropriate plug-in is not available:

C	: C language.
CC	: C++ language.
F90	: Fortran 90 language.
F95	: Fortran 95 language.

<UNIT-BODY> </UNIT-BODY>

The body of the full SPC-XML document contains declarations of processing units (processes and functions) and contention resources declarations. Allowed tags inside the program body are: CODE, PROCESS, FUNCTION, INCLUDE and RESOURCES.

<CODE> </CODE>

It should contain *cdata* which is not parsed by an SPC-XML front-end, but copied into the target language file. This tag may appear to define the body of a sequential function written in the native/sequential language or alone in the BODY of the program. In this last case the purpose of the CODE tag is to allow the programmer the inclusion of any specific sequential language instructions to initialize the program. For example, to add the `include` statements needed on a C language program.

<INCLUDE> </INCLUDE>

This tag must contain a valid filename on the system containing an SPC-XML document. This tag forces the inclusion of the BODY inside the other SPC-XML document before parsing. Nested inclusion should be supported by the front-end. An SPC-XML document is included only once even if it is several times included.

It is suggested that the DESCRIPTION of the other SPC-XML document should be included. Information enough to find the original filename and line numbers should also be added to processes and functions description tags in order to generate appropriate error messages during following compilation phases.

A.2 Processes and functions

<PROCESS name=""> </PROCESS>

It contains a named SPC-XML process definition. Name must be unique along all the process and function definitions on the document, and other included documents. The special name `main` identifies the process which is started when the program begins to run.

A process definition includes one PARAMETERS, one LOCAL and one BODY tag. A PROCESS tag may also include an optional USES tag to define the contention resources needed.

<BODY> </BODY>

The body of a process. It contains the sequential/parallel control structure of this process, including other processes and function invocations. Allowed tags inside a process body are: CALL, PAR, LOOP, IF, REPEAT and WHILE tags.

<FUNCTION name="" [workload=""]> </FUNCTION>

It contains a named function definition written on the native/target sequential language. Name must be unique along all the process and function definitions on the document, and other included documents.

A process definition includes one PARAMETERS and one CODE tag with the function body. The CODE tag contains the function definition in the native/target language. The input/output interface must be consistent with the PARAMETERS tag definitions.

The `workload` attribute is optional, and if it appears, it must contain an number representing the estimated mean workload of the sequential function. It may be used by the compiler for workload estimation and optimizations.

A function tag may also include an optional USES tag to define the contention resources needed.

<PARAMETERS> </PARAMETERS>

This tag includes a collection of parameter declarations using the IN, OUT, INOUT tags.

<IN name="" basetype="" [dim=""] />

This tag declares an *input* parameter with the given name. The `basetype` attribute contains a valid type on the native/target language. If programmer

data types definitions are previously included through a CODE tag they can be used as a basetype. The SPC-XML front-end does not verify the existence or proper declaration of the data types. However, it should use the basetype names to check type consistency on process and function calls.
The dim attribute is optional. If it appears, its value is a local variable name in which the integer dimensions of the real parameter are stored during the call of the function or process.
If multidimensional variables are used, the dim attribute is an SPC-XML variable or array with enough elements to contain all the values of the dimensions.

<OUT name="" basetype="" [dim=""] />
This tag declares an *output* parameter with the given name. All the details previously discussed about the IN tag applies to the OUT tag.

<INOUT name="" basetype="" [dim=""] />
This tag declares an *input/output* parameter with the given name. All the details previously discussed about the IN tag applies to the INOUT tag.

A.3 SPC-XML variables

<LOCAL> </LOCAL>
This tag includes a collection of local variable declarations for a process, using the VAR tag.

<VAR name="" basetype="" dim="" [shared=""] />
A variable declaration with a given name. For the basetype attribute see the IN tag declaration. The attribute `dim` is compulsory and must be a number (if only one-dimensional array variables are supported), or a list of numbers separated by colons (if multi-dimensional variable arrays are supported).
Single variables must be defined as one-dimensional arrays with only one element (`dim="1"`).
The optional attribute `shared` has a default value of "false". If its value is "true", it indicates to the compiler that it is safe to pass this variable to other processes using shared memory instead of sending a copy of the data (see memory model in section 2.4).

A.4 Invocation of functions and processes

<CALL name="" [exclusive=""]> </CALL>
A call tag represents an invocation of the process or function identified by the `name` attribute.
A CALL tag contains PARAM tags specifying the real parameters to be used as the formal parameters of the function or process.
The optional `exclusive` attribute has a value of `null`. If its value is a name, the invocation is mutual exclusive with any other invocation of this process which has an exclusive attribute with the same name. (We should consider for consistency checks to force the programmer to include declarations for valid mutual exclusion names in the LOCAL section).

<PARAM value="" [target=""] />

This tag represents a parameter substitution. A valid SPC-XML variable name or a reference to an element of a local overlap variable must appear on the `value` attribute.

The `target` attribute is optional. If it appears it indicates the name of the target formal parameter to which the substitution applies. If it is not present, the substitution applies to the first formal parameter not named in a target attribute that has not been yet substituted.

A.5 Data-layouts: Overlaps

<VAR-OVERLAP srcVar="" name="" layout="" />

This tag is a declaration statement which creates and associates the indexed overlapping-variable elements with portions of the source variable, using the `layout` attribute to select the shape and sizes of the overlapping sections. This attribute is a Fortran90 style sub-array specification with colon notation (e.g. `layout="[1:3][10:100:2]"` means a subarray of the `srcVar` with only its first to third rows and only elements in its even columns from 10 to 100). The layout specification may use the following macro specifications:

#P The total number of processors available in this process.

#i The value of the index of the overlap variable.

#ˆ The first index value in the original array.

#$ The last index value in the original array.

#<name> The name of a partition module to compute the layout. Partition modules should be provided by the compiler framework or supplied by the programmer.

A.6 Parallel control structures

<PAR [p=""]> </PAR>

This tag represents an spawning of `n` tasks which may be executed in parallel. Its content determines the exact code to be executed in each parallel branch. The only allowed tags inside a PAR tag are PARBLOCK tags.

The `p` attribute is optional. If it does not appear, the number of parallel branches should be deduced from the branches specifications inside the tag. The PAR tag may also have the special macro value (`#P`) for the `n` attribute. This value means that the number of parallel processes spawned is the same as processors available to this part of the computation (process).

<PARBLOCK [index=""]> </PARBLOCK>

This tag may only appear inside a PAR tag. It defines the behavior of one of the parallel branches spawned. It may contain CALL, LOOP, PAR, IF, REPEAT or WHILE tags.

The `index` attribute is optional. If it appears its content must be an integer number between 0 and $n - 1$, being n the content of the `n` attribute of the PAR tag.

When the PAR tag has an `n` attribute, one and only one of the PARBLOCK tags inside may have an special value for the `index` attribute. This value is a star `"*"`, representing a *wildcard* branch. The content of the wildcard branch tag is used for as many branches as n minus the number of other PARBLOCK tags inside the PAR. Inside the wildcard branch of a PAR tag, the programmer may use the special index macro `#i`, which has the value of the branch index.

A.7 Sequential control flow structures

<LOOP n=""> </LOOP>

This tag represents an `n` times sequential repetition of its content. The `n` attribute may be an integer number or an SPC-XML variable with dimension 1, which basetype may be used in the native/target language as an integer or loop limit.

The programmer must remember that in the first case the synchronization structure of the program is static and in the second case dynamic.

<IF> </IF>

This tag represents conditional execution of one tag inside. The only tags allowed inside an IF tag are a compulsory CONDITION tag, one THEN tag and one optional ELSE tag.

The presence of an IF tag typically implies a dynamic synchronization structure program. However, the parser/compiler may apply transformations to eliminate the dynamic structure (as IF reductions).

<CONDITION> </CONDITION>

The condition inside this tag is written in the native/target language, and it may use SPC-XML variable references that must be translated properly by the parser.

<THEN> </THEN>

The code to execute when the condition of an IF tag is evaluated to true. It may contain CALL, LOOP, PAR, IF, REPEAT or WHILE tags.

<ELSE> </ELSE>

The code to execute when the condition of an IF tag is evaluated to false. It may contain CALL, LOOP, PAR, IF, REPEAT or WHILE tags.

<WHILE> </WHILE>

This tag represents iterative execution of the inside tags. The compulsory CONDITION tag inside is evaluated before beginning each execution. The iteration continues *while* the condition is evaluated to true.

A WHILE tag may also contains CALL, LOOP, PAR, IF, REPEAT or WHILE tags.

The presence of a WHILE tag typically implies a dynamic synchronization structure program. However, the parser/compiler may apply transformations to eliminate the dynamic structure.

<REPEAT> </REPEAT>

This tag represents iterative execution of the inside tags. The compulsory CONDITION tag inside is evaluated after ending each execution. The iteration continues *until* the condition is evaluated to true.

A REPEAT tag may also contains CALL, LOOP, PAR, IF, REPEAT or WHILE tags.
The presence of a REPEAT tag typically implies a dynamic synchronization structure program. However, the parser/compiler may apply transformations to eliminate the dynamic structure.

A.8 Global named resources (optional)

<RESOURCES> </RESOURCES>
It contains a collection of RESOURCE tags, defining a collection of resources to content for.

<RESOURCE name="" [units=""] />
A contention resource definition, including a name and an optional number of units (1 by default).

<USES> </USES>
This tag contains a collection of USE tags.

<USE name="" [units=""] />
Each USE tag defines the name and units of a contention resource that a function or process must lock to proceed. The `units` attribute is optional, with a default value of 1.
The resources locking and unlocking operation must be atomic for all the resources inside the USES tag.

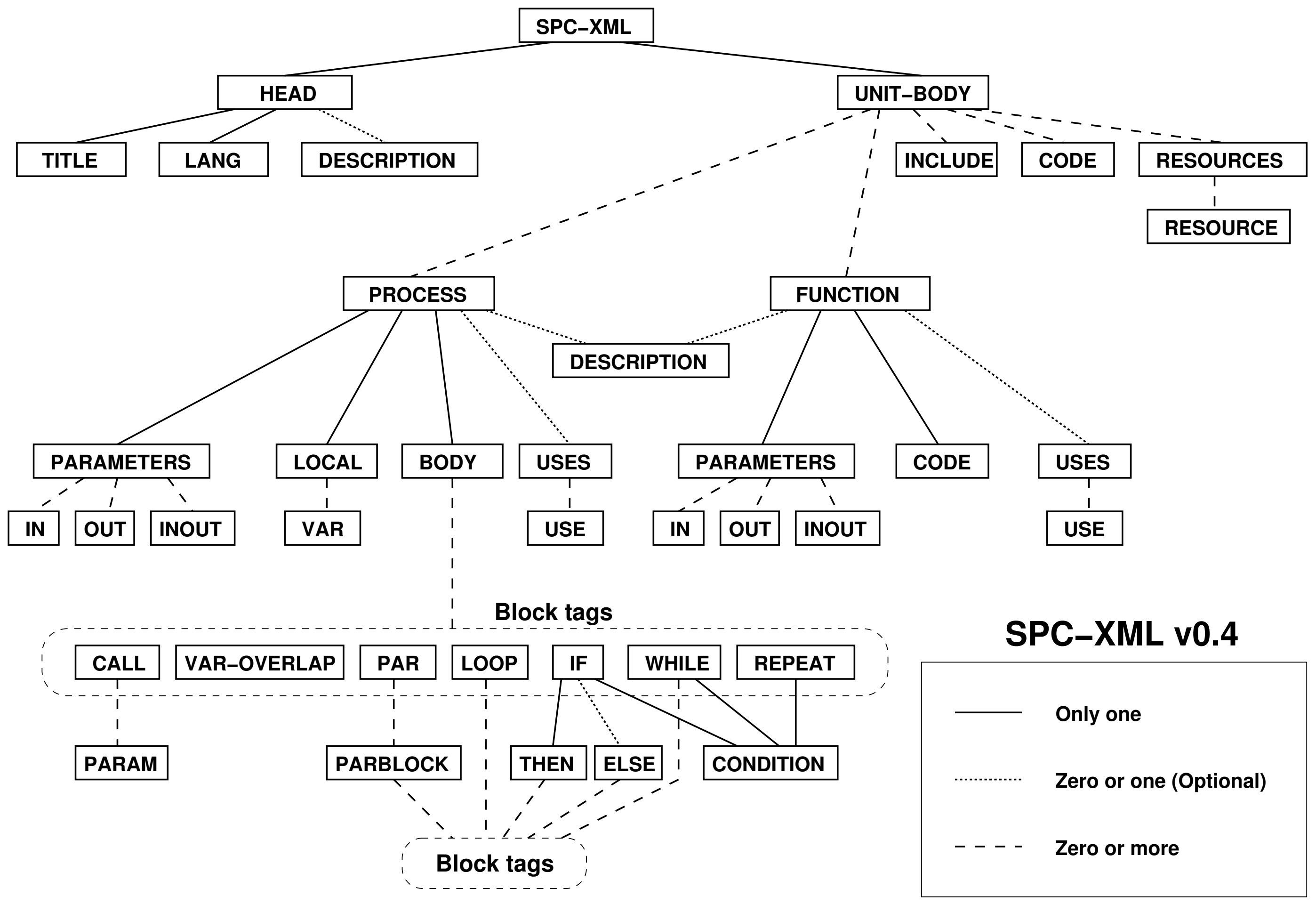

Fig. 1. SPC-XML document structure

B DTD (Data Type Definition)

```
<?xml version="1.0"?>
<!DOCTYPE SPC-XML [

<!ELEMENT SPC-XML (HEAD, UNIT-BODY) >
    <!ATTLIST SPC-XML version CDATA #FIXED "0.4">

<!ELEMENT HEAD (TITLE, LANG, DESCRIPTION?) >
<!ELEMENT TITLE (#PCDATA) >
<!ELEMENT LANG EMPTY >
    <!ATTLIST LANG name (C|CC|F90|F95) #REQUIRED >

<!ELEMENT DESCRIPTION (#PCDATA) >

<!ELEMENT UNIT-BODY ((INCLUDE|CODE|RESOURCES|PROCESS|FUNCTION)*) >
<!ELEMENT INCLUDE (#PCDATA) >
<!ELEMENT CODE (#PCDATA) >

<!ELEMENT RESOURCES (RESOURCE*) >
<!ELEMENT RESOURCE EMPTY >
    <!ATTLIST RESOURCE name CDATA #REQUIRED >
    <!ATTLIST RESOURCE units CDATA "1" >
<!ELEMENT USES (USE*) >
    <!ATTLIST USES name CDATA #REQUIRED >
    <!ATTLIST USES units CDATA "1" >

<!ELEMENT PROCESS (DESCRIPTION?,USES?,PARAMETERS,BODY) >
    <!ATTLIST PROCESS name ID #REQUIRED >

<!ELEMENT FUNCTION (DESCRIPTION?,USES?,PARAMETERS,CODE) >
    <!ATTLIST FUNCTION name ID #REQUIRED >
    <!ATTLIST FUNCTION workload CDATA #IMPLIED >

<!ELEMENT PARAMETERS (IN|OUT|INOUT)* >
<!ELEMENT IN EMPTY >
    <!ATTLIST IN name CDATA #REQUIRED >
    <!ATTLIST IN basetype CDATA #REQUIRED >
    <!ATTLIST IN dim CDATA #IMPLIED >
<!ELEMENT OUT EMPTY >
    <!ATTLIST OUT name CDATA #REQUIRED >
    <!ATTLIST OUT basetype CDATA #REQUIRED >
    <!ATTLIST OUT dim CDATA #IMPLIED >
<!ELEMENT INOUT EMPTY >
    <!ATTLIST INOUT name CDATA #REQUIRED >
    <!ATTLIST INOUT basetype CDATA #REQUIRED >
    <!ATTLIST INOUT dim CDATA #IMPLIED >

<!ELEMENT LOCAL (VAR*) >
<!ELEMENT VAR EMPTY >
    <!ATTLIST VAR name CDATA #REQUIRED >
    <!ATTLIST VAR basetype CDATA #REQUIRED >
    <!ATTLIST VAR dim CDATA #REQUIRED >
    <!ATTLIST VAR shared (yes|no) "no" >

<!ELEMENT BODY (CALL|VAR-OVERLAP|PAR|LOOP|IF|WHILE|REPEAT)* >

<!ELEMENT CALL (PARAM*) >
    <!ATTLIST CALL name CDATA #REQUIRED >
    <!-- Cannot be an IDREF to allow calls to elements in other documents -->
    <!ATTLIST CALL exclusive CDATA #IMPLIED >

<!ELEMENT PARAM EMPTY >
    <!ATTLIST PARAM value CDATA #REQUIRED >
    <!ATTLIST PARAM target CDATA #IMPLIED >

<!ELEMENT VAR-OVERLAP EMPTY >
    <!ATTLIST VAR-OVERLAP srcVar CDATA #REQUIRED >
    <!ATTLIST VAR-OVERLAP name CDATA #REQUIRED >
    <!ATTLIST VAR-OVERLAP layour CDATA #IMPLIED >

<!ELEMENT PAR (PARBLOCK*) >
    <!ATTLIST PAR p CDATA #IMPLIED >
<!ELEMENT PARBLOCK (CALL|VAR-OVERLAP|PAR|LOOP|IF|WHILE|REPEAT)* >
    <!ATTLIST PARBLOCK index CDATA #IMPLIED >

<!ELEMENT LOOP (CALL|VAR-OVERLAP|PAR|LOOP|IF|WHILE|REPEAT)* >
    <!ATTLIST LOOP n CDATA #REQUIRED >

<!ELEMENT IF (CONDITION,THEN,ELSE?) >
<!ELEMENT THEN (CALL|VAR-OVERLAP|PAR|LOOP|IF|WHILE|REPEAT)* >
<!ELEMENT ELSE (CALL|VAR-OVERLAP|PAR|LOOP|IF|WHILE|REPEAT)* >

<!ELEMENT WHILE (CONDITION,(CALL|VAR-OVERLAP|PAR|LOOP|IF|WHILE|REPEAT)*) >
<!ELEMENT REPEAT (CONDITION,(CALL|VAR-OVERLAP|PAR|LOOP|IF|WHILE|REPEAT)*) >

]>
```

C Previous version: Transitional documentation

The changes from version v0.3 to v0.4 include minor tag renaming and new unified syntax for the inclusion of dynamic data-layouts and partition modules extensions (see the description of the new VAR-OVERLAP tag).

C.1 Tags renamed or substituted from v0.3

- OVERLAP element is substituted by VAR-OVERLAP. The VAR-OVERLAP element is not a processing statement, as the previous OVERLAP tag. It is a declaration statement and may only appear inside the LOCAL element.
- BRANCH element is renamed as PARBLOCK.
- BODY element only exists now inside a PROCESS element.
- BODY element inside SPC-XML element is renamed as UNIT-BODY.
- Attribute shared of the VAR element is now a boolean value "true" or "false"; instead of using as values the words "yes", "no".
- Tag MAIN has been eliminated. Now, a PROCESS element which have a name attribute with value *"main"* is considered the process which is run to begin the program's execution.

C.2 Version 0.3.1: Dynamic data-layouts

This is an obsolete transitional documentation. Version v0.4 has included most of these features in the new VAR-OVERLAP tag.

Dynamic data-layouts: Dynamic overlaps

<OVERLAP srcVar="" overlapVar="" layout=""/>
This form of the OVERLAP tag is thought for dynamic even partitions. The dynamic back-end of the compiler should create the appropriate target code to create an overlap of the source variable accordingly with the source variable dimensions, the number of processors/threads available to this part of the computation and the `layout` type.
Partitions type are well-known data layout forms:

blockR : Each part of the overlap has the same size (approximately if not even) and its data pieces are consecutive.
If multidimensional variables are used, the *first dimension (rows)* is the one chosen to split in blocks (more efficient for row oriented memory layouts, like C language).

blockC : Each part of the overlap has the same size (approximately if not even) and its data pieces are consecutive.
If multidimensional variables are used, the *last dimension (columns)* is the one chosen to split in blocks (more efficient for column oriented memory layouts, like Fortran language).

strideR : Each partition is a collection of rows (first dimension elements) with stride equal to the number of available processors/threads.

strideC : Each partition is a collection of columns (last dimension elements) with stride equal to the number of available processors/threads.

Other more complex layout types (as tiles) may be defined (and included preferred as compiler plug-ins) in the future.

The programmer may use a dimensionless overlap variable. The dimension of the overlap variable is computed accordingly with the number of processors available. The macro **SPClast** may be used to refer to the last valid index of an overlap.

<BORDER borderVar="" layoutVar="" type="" size=""/>

This last form of overlapping is thought to create the appropriate layout overlaps needed to communicate border information of a previous data-layout.

The `borderVar` attribute is the name of a new overlap variable in which to store the border shape information. The `layout` attribute is the name of the overlapping variable associated with a previous layout, using an OVERLAP tag.

The new border overlap will contain as many parts as the original layout, and each part will contain only a subset (border) of the same indexed part. The border position is determined by the `type` attribute, and the `size` attribute determines how many elements (on the appropriate dimension) are included in the border.

Valid border types are:

first : First elements of a block partition (rows or columns).

last : Last elements of a block partition (rows or columns).

More types could be defined (and included prefer ed as compiler plug-ins) in the future.

References

1. J.A. Caminero-Granja. *SPCC, un protipo de un compilador de SPC-XML.* M.sc. thesis, E.T.S. de Ingeniería Informática, Sep 2004.
2. C. Lázaro de la Osa and P. Collazos Fernández. *Técnicas de compilación para lenguajes de paralelismo Anidado.* M.sc. thesis, E.T.S. de Ingeniería Informática, Sep 2002.
3. D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, Feb 1992.
4. A.J.C. van Gemund. The importance of synchronization structure in parallel program optimization. In *Proc.* 11th *ACM ICS*, pages 164–171, Vienna, Jul 1997.
5. R. Portales-Fernández. *Compilador estático-dinámico de SPC-XML (Estudio e implementación).* M.sc. thesis, E.T.S. de Ingeniería Informática, Sep 2003.