An Architecture Description Language for coarse-grained Reconfigurable Arrays

Julio Oliveira Filho Stephan Masekowsky
Thomas Schweizer Wolfgang Rosenstiel
Department of Computer Engineering, University of Tübingen
Sand 13, 72076 Tübingen, Germany
crc@informatik.uni-tuebingen.de

ABSTRACT

The high degree of freedom in the design of coarse-grained reconfigurable arrays imposes new challenges to their description and modeling. We introduce an architecture description language targeted to describe coarse-grained reconfigurable architecture templates. It comprises innovative key features to allow fast modeling and analysis of such architectures, namely: representation of processing element array (ir)regularities, and flexible and concise description of interconnection network. We demonstrate that the proposed language enables a formal validation of the described template. Finally, we show how we automatically generate a SystemC based simulator of the described architecture. Our results suggest that the semantic and technical innovations of the proposed architecture description language may positively impact on the productivity of the design phase.

1. INTRODUCTION

Coarse-grained reconfigurable architectures(CGRAs) have increasingly gained the attention of academia [9][12][18] and industry [19] in recent years. In comparison to FPGAs, they reduce area, power, and configuration time. They also offer a more predictable timing, less configuration storage space, and an easier integration with a processor. Such advantages are obtained at cost of post-production flexibility, and thus these architectures must be targeted earlier at design phase toward a set of applications.

As these systems grow in complexity, their design phase demands the use of higher abstraction levels to deal with time-to-market and quality pressures. Indeed, such architectures may vary profoundly in the granularity of its processing elements(PEs), structure of the network interconnection, memory architecture, among other aspects. The result is a very large and complex design space. Moreover, the success of a product also remains in the capability of co-generating tools such as compilers and simulators. We compare this situation to the development of application-specific processors, where a successful approach was to use architecture description languages (ADLs). ADLs offer a formal description of architectures and its properties at a higher abstraction level. Their semantics are simple and definite enough to allow analysis, verification, and possibly automatic tool generation.

The purpose of this study is to investigate an architecture description language for modeling coarse-grained reconfigurable array architectures. Our contributions in this paper are:

- We discuss key features and technical innovations necessary for an ADL aimed for the description of CGRAs. In short, description of PEs functionality and structure, capability to capture the array composition, and flexible and concise description of interconnection network.
- We introduce an ADL designed to describe CGRA parameterizable templates. We show through some examples its syntax, semantic, and technical innovations to deal with the previously discussed challenges.
- We demonstrate the impact of our ADL on the development of realistic architectures. First, we show it provides a natural rationale to formal verification of the described interconnection network structure. Second, we detail how the ADL was used to generate a SystemC based simulator of the described architecture.

We produce short, easy-to-understand descriptions when compared to SystemC and XML based approaches. This paper is organised as follows. In section 2, we discuss the design flow of CGRAs and we present recent work on that area that follows a similar approach. In section 3, we present our ADL. Our results are grouped in sections 4.1 and 4.2, where we show how our ADL may positively impact on the productivity of the design phase of CGRAs.

2. DESIGN OF COARSE-GRAINED RECONFIGURABLE ARRAYS

We discuss the design phase of CGRAs using the CRC model [16]. We consider the development of CGRAs follows the three steps depicted in Figure 1. At the first step, the architecture is conceived very generically as an array of PEs surrounded by a network interconnection and memory blocks. At a second step, the designer captures this concept by means of a parameterizable architecture template. The

template describes the granularity, type and disposition of PEs, the possible network interconnections and the structure for the memory architecture. Templates are still being a very flexible platform. Modifications are allowed through parameters that may vary within a certain range of values. Parameters determine qualitatively or quantitatively some aspect of the architecture. Common examples are the width and height of the array, number of reconfiguration contexts, number of internal registers within the PE, and interconnection network. At the third step, one architecture instance can be generated assigning a value to each parameter. That instance may then be synthesized, evaluated and/or simulated.

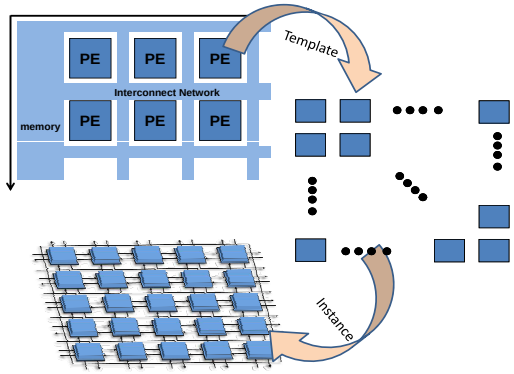

Figure 1: General design flow for coarse grained reconfigurable architectures

One bottleneck in this design flow is on the description of the template. Several groups [9][18] use a hardware description language(HDL), such as VHDL or Verilog. That has the advantage to ease synthesis at a later stage of the development. However, it presents several drawbacks. First, HDL-based templates are hard to write and modify. For example, including a new module may imply several distributed modifications in the description of the interconnection network and how it scales with different parameter values. Common HDL features, like the VHDL `generate`, are too unclear to that purpose. Second, it is difficult to extract information from the code in order to perform analysis, formal verification, or derive tools, such as a compiler or simulator. Both problems occur because HDLs are meant to be very general and have a too low abstraction level.

We make an analogy to actual advances on the design phase of application specific processors(ASIPs). The constant increase in system complexity and the pressure for design productivity pushed ASIP designers to work at higher abstraction levels. Architecture and system description languages were proposed [8], that offer a set of properties to deal efficiently with the previously mentioned problems.

Our proposal is to investigate the design of an ADL able to deal with the challenges of CGRA design phase. In the following, we briefly discuss the state-of-the-art of ADLs and position our contribution.

2.1 Related Work

Dutt [15] proposes a classification of ADLs based on two different criteria: (a) the type of description (structural or behavioral), and (2) the purpose they should accomplish. Figure 2 depicts some ADLs grouped according to this classification. MIMOLA [2] is a very powerful ADL at register transfer level. It was developed to support simulation, compiler generation and synthesis. However, its strength resides on describing the structure. Behavior is described as a secondary aspect apart. Contrarily, nML[4] and ISDL[6] are centered on describing the instruction set of a processor, and thus are behavior oriented. They support the generation of very efficient simulators and compilers but are constrained toward synthesis. Some ADLs, such as AIDL[14], describe the datapath structure only partially. The objective of AIDL is to validate the behavior of super-scalar pipelines. Most of the ADLs, however, are able to describe structure and behavior. Well known representants which partly inspired our work are EXPRESSION[7], LISA[10], and ArchC[1]. All mentioned languages are presently in a very mature state and have positively demonstrated their impact on the design of application specific instruction set processors.

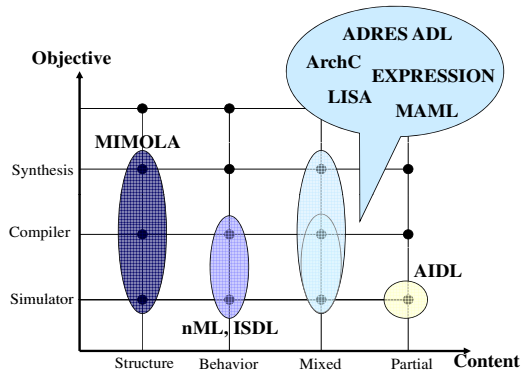

Figure 2: Classification of Architecture Description Languages

However, they are not adequate to describe CGRAs. They lack constructs to efficiently deal with (ir)regularities of array structures, scalability through parameterization, and interconnection network. We detail these challenges in the next section. First, we would like to present some research work similar to our approach.

IMEC developed ADRES, a coarse-grained reconfigurable array tightly coupled to a VLIW processor core. They describe in several publications [3] [13] the use of an XML-based description of the architecture. From this description they generate VHDL and ESTEREL code files used for synthesis and simulation, respectively. Unfortunately, this XML notation has never been published, and its characteristics and potentialities are not clear. The machine markup language MAML [5], initially created to describe parallel processor architectures, was recently extended to describe the so called weakly-programmable arrays (WPPAs) [11]. MAML is an XML based language and its extension addresses several of the challenges we discuss here. In order to allow a comparison with our approach, we will discuss common developments along the remainder of this paper. The strongest drawback of MAML is the readability. Description files are generally overflooded with complex markup tags. That makes these files difficult to read, write, and modify without a software tool. To the best of our knowledge, we are the first group to propose a non-XML based language targeted to the description of CGRAs.

2.2 Challenges to a CGRA targetted ADL

Our ADL was designed considering the general four items discussed in section 2. The designer hierarchically describes the architecture using constructs that express common basic coarse grained modules, such as registers, multiplexers, finite state machine, and their interconnections. That allows highly abstract descriptions and keeps a clear and definite semantic. We chose not to use XML to keep the code easy to read and modify. Analysis and toolkit generation are simplified because implementation details are hidden in the ADL semantic. Additionally, the proposed ADL addresses several specific demands when describing CGRAs:

Processing elements The internal structure of PEs may be independently described. Behavior is embedded in the organization of the internal datapath and in functional units (FUs). FUs are modules with customizable behavior. Because we use a PE-centric approach on our descriptions, non-PE elements such as shared buses and central register files may be modeled as processing elements without functional units. We detail the description of processing elements in section 3.2.

Homogeneous and Heterogeneous arrays CGRA arrays may be composed of very different PEs. For example, it is common to provide multipliers only in part of the PEs in order to optimize area and energy. Or yet, border PEs provide usually extra resources to communicate with the external environment. Our ADL uses a Matlab-based notation to describe the disposition of PEs over the array according to their types. It is a very concise, intuitive and scalable way to describe regular as well as irregular PE compositions. More details in section 3.3.

Scalability Our language provides a very flexible syntax for the connection between modules. It fully enables scalable structures such as modules with a variable number of input and output ports. Section 4.1 shows how scalable architecture templates may be formally validated.

Network interconnection Description of network interconnection is a highlight of our ADL. We introduce the idea of connection rules. Instead of describing direct connection between PEs, the designer describes a rule on how to connect it. It is especialy interesting to capture the regular and scalable characteristic of interconnection networks in CGRAs.

Multi-Context Reconfigurability Some CGRAs employ the so called multi-context reconfigurability. The reconfiguration data is stored in special units called context memories. Reconfiguration takes place very fast (within one clock cycle) by switching the entries in the context memory. Because this feature is increasingly adopted in actual CGRAs, we built constructs to support the description of context memories.

In the next section, we describe our ADL with focus on the previously mentioned aspects.

3. THE CRC ARCHITECTURE DESCRIPTION LANGUAGE

We developed a simple and intuitive ADL to describe coarse-grained array-based architecture templates. The ADL was designed generic enough to allow the description of a broad palette of CGRAs. However, for the sake of simplicity, we introduce it here using the CRC-Model. Our ADL consists of three main sections: The `PARAMETER`, the `PE` and the `ARCH` sections. First, we introduce the `PARAMETER` and the `PE` sections using the simple example depicted in Figure 3.

```
1. PARAMETER REGSETSIZE IN [8,16,32..64];
2. PE {
3.  INPORT(8),OUTPORT(8);
4.  MUX dinMux1;
5.  REG dataRegisterSet(REGSETSIZE);
6.  FSM fsm(4);
7.  CONTEXTMEMORY contextMemory(4);
8.  FU myFU(unsigned_add,unsigned_sub);
9.  CONNECTION {
10.  dinMux1(INPORT[3..0],
11.           dataRegisterSet[0..REGSETSIZE-1],
12.           contextMemory[0]);
13.  //...
14.  }
15. } myPe;
```

Figure 3: A simple example. PARAMETER(1.) and PE sections(2.-15.).

3.1 The Parameter section

The first step when designing CGRAs with the CRC-Model is capturing the architecture as a parameterized model: the architecture template. Many characteristics of the architecture are not yet fixed in the template. Parameters are used to describe them or how they may vary. Further in the design process, an architecture instance is obtained assigning values to each parameter. To allow flexible and scalable templates, our ADL includes mechanisms to parameterize an architecture description.

In the `PARAMETER` section, we can declare an arbitrary number of parameters with their associated values. The values for the parameters can be single numbers or ranges given in any order. Figure 3, line 1, depicts an example of parameter declaration. The parameter `REGSETSIZE` may assume the values 8, 16, or any integer value in between 32 and 64. Declaring the possible values is necessary for validation purposes (see section 4.1). Parameters can be used as variables when declaring modules and interconnections. For example, line 5 declares a register set whose size is defined by the value attributed to `REGSETSIZE`. The same parameter is used to control the connections between the register set and the multiplexer `dinMux1` (line 10-12).

3.2 The PE section

In the `PE` section we describe the structure of a processing element. It consists of a list of declarations for the PE elements such as multiplexers, finite state machine, register set, functional units, context memories,and input and output ports. Several PEs can be described independently using different PE sections. A name is assigned to each PE for later identification. Lines 3 to 8 in Figure 3 declare that the PE `myPe` has 8 input and 8 output ports, 1 multiplexer, 1

parameterizable register set, one finite state machine with 4 states, a context memory with 4 context entries and one functional unit capable of unsigned addition/subtraction. The semantic for each one of these elements is discussed in the following:

Ports The ports of a processing element must be declared in order to define the interface between the PE and the array. Input and output ports are declared using the key words `INPORT` and `OUTPORT`, respectively, and the total number of ports (line 3).

Multiplexer Multiplexers are common components in reconfigurable datapaths within a PE. Our ADL semantic assumes a multiplexer to have several input ports, one output port and one select port. One of the input port signals is transferred to the single output port according to the signal of the select port. We declare a multiplexer in our ADL using the keyword MUX and an identifier (line 4). The actual number of its input ports will be determined based on the PE `connection` section. In the example in figure 3, the number of input ports depends on the number of registers present on the register set.

Register Set A register set describes a module with one input port, one address port, and r-output ports, where r is the number of registers in the set. The keyword `REG` is used to declare a register set. The number of registers in the set is assigned at the declaration. It may be a single value, like in `REG myRegSet(2)`, or a parameter like in line 5.

Finite State Machine The keyword `FSM` is used to declare a finite state machine module. We assume the FSM to be a clocked hardware block with one input signal (condition port) and one output signal. At each clock cycle, the state of the output signal is determined depending on the signal present at the condition port. Additionally, it is necessary to determine the maximal number of internal states used in the FSM. This attribute can also be determined through a parameter. For example, the code `FSM fsm(FSMSIZE)` declares a FSM with `FSMSIZE` states. The behavior of the finite state machine is not defined at this stage.

Context memory In our ADL, a context memory (see line 7) is an innovative construct. Each entry in this memory model stores one context. Each context determines the control signals of other hardware modules such as multiplexer select signals, register addresses, and FU operation codes. It has one address input port, usually connected to the output port of the FSM. The idea is that at each clock cycle, the FSM switches to a new state and outputs the corresponding context address. That address selects one entry in the context memory, which outputs the signals to control the datapath. The number of the output ports in the context memory is determined in the PE `connection` section. The number of entries in the context memory is determined by the designer and can be parameterized as well. Embedding the concept of a context memory within our ADL is a key feature to facilitate the description of multi-context reconfigurable architectures. For example, we differentiate here from MAML, where the context memory must be explicitly described using a combination of program counter, instruction memory and instruction decoder.

Functional Unit The FU is the most flexible module in our ADL. It models a hardware module with an operation select signal port and an arbitrary number of data input and output ports. During the FU operation, one specific operation is chosen through the select signal port. The corresponding processing algorithm is called using the data present in input ports as parameters. The result is then presented in the output ports. To declare an FU, the designer lists all the operations that the functional unit should support (line 8). The behavior of each listed operation is determined later by the designer. Actually, we write a C++ routine outside of the ADL description for each operation reference. That is not very efficient and would be more interesting to maintain the same language while describing structure and behavior. Because of that, we concentrate our actual efforts on integrating the behavioral description within the ADL.

A diagram view of the PE described in Figure 3 may be seen in Figure 4. For sake of simplicity, the depicted PE is not complete and corresponds only to the described code. After declaring all components of a PE, we describe their interconnections using the PE `connection` section.

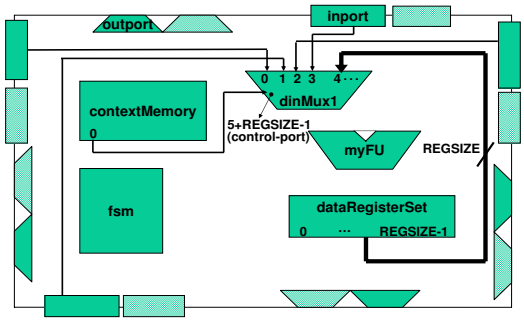

Figure 4: Processing Element

3.2.1 *The connection section*

Connections between elements are described using a simple connect-by-position mechanism. For each PE-element, we provide a comma separated list. The entries in the list correspond to the input ports of the element in ascending order. The general scheme is the following:

`sinkID(sourceID`$_1$`[outPort], sourceID`$_2$`[outPort],...)`

where `sinkID` and `sourceID`$_x$ are identifiers declared previously within this PE. `outPort` may be a number, a parameter, or an interval. In each case, it designates the output ports of `sourceID`$_x$ to be connected in that entry. A number is used when the output port is the same for all architecture instances generated from this template. A parameter describes an output port that will be fixed only at instantiation phase. An interval is used to connect several output ports in a sequence.

The example in Figure 3 (lines 10-12) declares the connections to the module `dinMux1`. You can see the described PE

in figure 4. According to this description, input ports 0, 1, 2 and 3 of `dinMux1` are connected to input ports 3,2,1 and 0 of the PE, respectively. The output ports 0 to (`REGSETSIZE`−1) of `dataRegisterSet` are connected. Finally, the output 0 of the context memory is connected to the last input port of the multiplexer. Note that the number of necessary input ports of `dinMux1` automatically considers the value of the parameter `REGSETSIZE`. Contrary to VHDL and Verilog style, we explicitly avoid the declaration of signals when describing connections. That is an advantage because it keeps the description small and easy to understand.

3.3 The ARCH section

In the `ARCH` section we arrange the previously declared PEs on the array and connect them to establish the interconnect network. The easiest way to describe an array is to write down its dimensions and assume that all the PEs are identical. The result will be a homogeneous array. However, as discussed in section 2.2, describing heterogeneous and irregular arrays is a growing practice. In fact, after CGRAs are specialized to a set of applications, they usually present a structure with different PEs organized in a somehow regular pattern. An efficient ADL should be able to express array irregularities as easily as it describes its regularities.

We developed a simple scheme to describe the array structure similar to the way MATLAB concatenates matrices. First, the designer captures local irregularities in the array by grouping possible heterogeneous PEs in basic blocks. Defined basic blocks may be then horizontally or vertically concatenated to form larger and more complex blocks. Finally, that pattern is replicated regularly using the ADL construct `ARRAY`. Figure 5 depicts that idea. `B` is a 2×2 basic block with two different PEs in a chessboard like arrangement. This pattern is concatenated with two other PEs of type 1 to form another block `D`. The unique restriction to the concatenation is that blocks must have a rectangular shape. For example, `C` is a bad formed basic block.

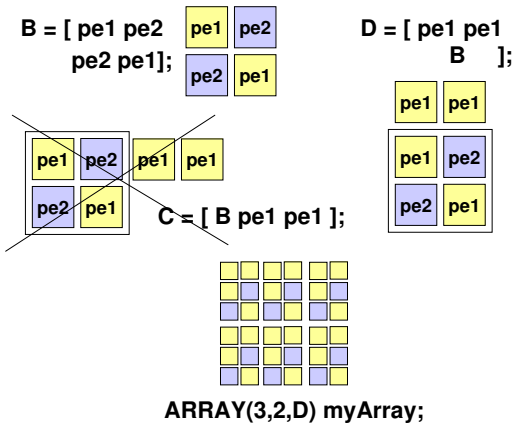

Figure 5: Describing the arrangements of PE in the array

A regular array for the CRC-Architecture can then be defined using the keyword ARRAY followed by three entries: the array width, the array height and the identifier of the block to be replicated. In Figure 5, `myArray` is a 3×2 array of `D` blocks. The `ARRAY` construct may be parameterized. For example, the command `ARRAY(width, height, D)` describes an array whose geometry is defined through the parameters `width` and `height`. Thus, the template has no fixed geometry, and all possible value combination for these parameters generate a different architecture instance. In our ADL, the `ARRAY` construct may be used an arbitrary number of times within the same description.

This MATLAB-like syntax allows a very fast, concise, intuitive, and definite way to describe array arrangements. Here, we can make another comparison to a similar approach in MAML language. MAML also efficiently captures the array arrangement and geometry using a mathematical formulation. They group PEs in so called domains, where each domain is represented using mathematical polytopes equations [11]. Their approach have an equal description capability as ours, however, the polytopes equations are much more difficult to understand, formulate, and modify.

3.3.1 *Interconnect network*

The interconnect network is one of the most variable and complex aspect of CGRAs. Nevertheless, they are critical to the overall architecture performance, area, and power consumption. We introduce an innovative concept to describe the interconnect network at array level. Instead of describing explicit connections between PEs ports, we partition the array into regions so that all PEs within one region have their input ports connected the same way. Then for each region, we write only once how the elements are to be connected. In order to illustrate this idea, we refer to Figure 8a. All internal PEs of this instance have their input ports connected directly to their adjacent neighbors. Hence, we must only describe that region (internal PEs) and its connection pattern (adjacent neighbors). We call *connection rule* to the complete set of regions that cover one array and their connection patterns.

Before discussing our connection rules in detail, we explain how we use it in a design. Consider the `ARCH` section described in Figure 6. After declaring `myArray`, connection rules are listed within the array `connection` section. Such rules are defined independently of the specific arrangement of the PEs in an array, and may or may not depend on its dimensions. Each rule is introduced with the keyword `RULE`, which is followed by the description of the rule (not shown) and a name for later identification. Then, one connection rule is assigned to the array (line 8). This binding indicates the rule that will be used when generating one architecture instance.

A connection rule has three parts. In the first, we partition the array into disjoint sub-sets of PEs, called regions, and describe the corresponding connection pattern for each region. The second part defines which PEs output ports are connected to the array output ports. After that, there may still be unconnected output ports. Those ports are dead ended signals, that are never read inside or outside the array. We list them in part three for the sake of completeness of a full description. That is important when validating the model, as we discuss in section 4.1.

In order to describe regions in an array, we defined a coordinate system. Assume an array of arbitrary width W and height H. All array positions (w, h) can be numbered from (0,0), in the upper left corner, to $(W-1, H-1)$ in the lower right corner of the array. Additionally, we may designate a position relative to the array limits. For exam-

```
1. ARCH {
2.   B=[ pe1 pe2 ; pe2 pe1];
3.   ARRAY(width,height,B) myArray;
4.   CONNECTION {
5.     RULE {
6.      PE IN (1:END-1,1:END-1)
7.          (REL_COORD(-1,0)[2],REL_COORD(0,1)[3],
8.           REL_COORD(1,0)[0],REL_COORD(0,-1)[1]);
9.      ...
10.     LOG {
11.       PE IN (END,:)[0];
12.     } nearest_neighbor;
13.    RULE {...}myRule;
14.    // myArray(myRule);
15.    myArray(nearest_neighbor);
16.  }
17.}
```

Figure 6: Example of an ARCH section

ple, (END$-w$,END$-h$), where END is a keyword that describes the last position in a row or column. To describe a set of positions using this coordinate system, we use a syntax and semantic similar to the MATLAB colon operator. The colon operator applies the following syntax:

```
(begin:step:end, begin:step:end)
```

There are two parts divided by a comma. The first part selects a set of rows and the second a set of columns. All positions (w, h) corresponding to the intersection of these two sets are selected. Considering this notation, there are some abbreviations for special cases. For example, instead of writing down (0:1:END,0) which selects the very first column of the array, we can also write (0:END,0) or even (:,0). Another examples on how to choose (ir)regular sets of PEs are depicted in Figure 3.3.1. The first example selects all PEs in the fourth column. The second selects all PEs that are simultaneously in even columns and even rows. Finally, the internal PEs of an array may be selected as shown in the third example. One very important aspect is that these rules describe the same region

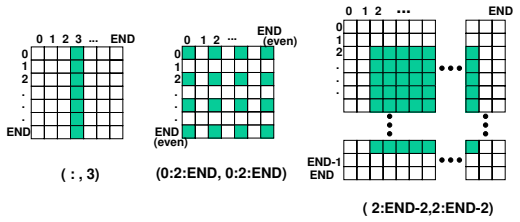

Figure 7: Regions representing subsets of PEs in the array

All PEs within a region become the same connection pattern to their input ports. Hence, we may group them in an abstract type PE^r. For each region, we write a list with a connect-by-position mechanism similar to the one used in the PE section. Each entry in this list represents input ports in increasing indexing order for the PE^r of this region. The example in Figure 6, lines 6 to 8, connects all internal PEs in an array in the following way. Input port 0 is connected to output port 2 of the adjacent PE in north. Input port 1 is connected to output port 3 of the adjacent PE in east. Input port 2 is connected to output port 0 of the adjacent PE in south. And input port 3 is connected to output port 1 of the adjacent PE in west. That connection is known as *nearest-neighbors*, and can be seen depicted in Figure 8a. There are also constructs to describe connection to array absolute coordinates(ABS_COORD), constants(CONST) and array input ports(INPUT). However, we omit them here.

The second part of a connection rule is a list of output ports of PE^r that will be connected to output ports of the array. We call that the LOG section. Line 11 in Figure 6 designates that all PEs in the last array row have their output port 0 connected to array output ports. After describing parts 1 and 2, there still may be PEs on the array whose output ports have not been connected. Those are unused ports that have to be closed/terminated. So the third part of a connection rule is a list, like the one in the LOG-section, of all those output ports of the PEs that have not been connected. This part starts with the keyword VOID to indicate those ports go to nowhere.

The *connection rules* introduced in this section are an innovative and powerful mechanism to describe interconnect networks for coarse-grained arrays. They are flexible, concise and scalable with the parameterization of the template. Moreover, they offer naturally a mathematical approach for validation of the described network. We discuss that in the next section.

4. RESULTS

4.1 Validation of templates

To guarantee correctness when generating Software-Toolkits such as simulator and compiler, the described template must be *coherent*. That means it must describe a correct architecture instance for all foreseen parameter value combinations. The objective of this section is to demonstrate that the proposed ADL allows a natural way to validate described templates. We present, as examples, some problems that may occur when describing the interconnect network. For each problem, we derive, directly from the language semantic, mathematical properties that must be verified.

We said earlier, that one connection rule defines a partition of the array in regions. That is so because if two regions are allowed to have a common position, the PE at this position would be connected using two different schemes. So, we may define a mathematical property that should be respected by a connection rule. We call Ω the set of all positions in the array, and $\omega_r \subset \Omega$ the subset of positions corresponding to region r. Then, for all connection rules R the following property must be true:

$$\forall r \in R, \bigcup \omega_r = \Omega \wedge \bigcap \omega_r = \emptyset$$

The definition of a connection rule may be flawless in the sense above, but its binding to an array may fail. That occurs, for example, when the PE assigned to a given position has a different number of input ports as described for the region that contains this position. For each PE p in the array, we define a function IN(p) = number of input ports of p. Accordingly, for each region r in the connection rule R, we define IN_r^R as the number of entries in its connection list.

Hence, a template is *coherent* only if for all generated architecture instance, the following property can be asserted:

$$\forall p \text{ in array}, \text{IN}(p) = \text{IN}_r^R$$

The third example deals with the parameterization of the template. If a connection rule is not correctly described, it is possible that for some parameter value combinations the interconnect network will be malformed. Consider the 4×4 template depicted in Figure 8. Suppose we declared "`PARAMETER NHOP in [0..2]`" in the parameter section, and that the described connection rule defines a down-forward-hop network to the internal PE region as follows:

```
PE IN (1:END-1,1:END-1)
    (REL_COORD(-1,-NHOP)[2],REL_COORD(0,1)[3],
     REL_COORD(1,0)[0],REL_COORD(0,-1)[1]);
```

When `NHOP` has value 0, we have a simple nearest-neighbor network, as depicted in Figure 8a. The origin of the connections from the upper row are shifted once, when the parameter `NHOP` has value 1. The result may be seen in Figure 8b. One problem occurs for `NHOP` equal to 2. In that case, the PEs in the second column cannot be properly connected because there are no described PEs in the resulting position.

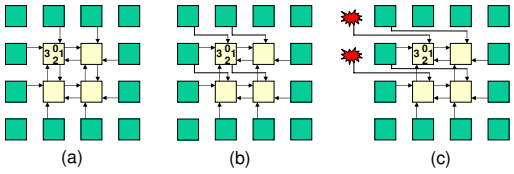

Figure 8: Instances of the Nearest-Neighbor-Down-Forward network

Such kind of problem may be verified observing the connection of output ports. Two conditions must simultaneously hold. First, the described coordinate for the source PE must be valid. Second, the PE in the described coordinate must have enough output ports. Now, we derived properties using the semantic in our ADL. We call c_e^r the coordinate described in entry e of the connection list of region r. If the entry is described as `INPUT` or `CONST`, then $c_e^r = (0,0)$ which describes always a valid position. Additionally, we define a function $\text{PORT}_r(e)$ as the output port declared in entry e of region r. For example, for the entry `(REL_COORD(3,2)[1]`, $c_e^r = (3,2)$ and $\text{PORT}_r(e) = 1$. We also define for all PE p a function $\text{OUT}(p)$ = number of output ports of p. When validating if a possible architecture instance is *coherent* the following property must be true:

$$\forall r \in R, \forall e \in r, c_e^r \in \Omega \wedge \text{OUT}(p) \geq \text{PORT}_r(e)$$

Although we do not cover here all the necessary properties to assure *coherence*, the three examples above show that our ADL offers a rational basis for validation and verification of the described model. We used this mathematical approach to implement these and other properties in a validation tool within the CRC design flow. The complete validation of a parameterized model considering 159744 possible parameter value combinations takes about 217 seconds.

4.2 Simulator Generation

Based on the proposed ADL, we automatically generate a cycle accurate simulator in SystemC. Figure 9 depicts the flow for the simulator generation. First, the ADL description of a template is parsed. During parsing, syntax and some semantic correctness is verified. When the parser ends, the validation unit proves the required properties of the model, as discussed in section 4.1. We allow the generation of a simulator only for *coherent* templates. If no inconsistencies were found, the generator is started.

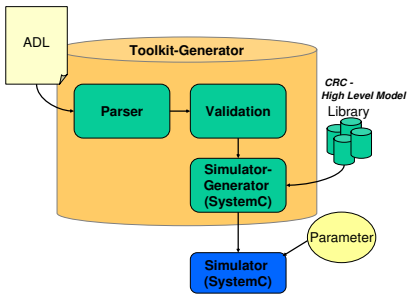

Figure 9: The CRC-ADL Toolkit Generator

We generate a simulator using a library of hardware modules written in SystemC. This library is called CRC High Level Model. It contains models that follow the semantic for all PE components present in the ADL, such as multiplexers, register set and context memory. It also provides connection mechanisms such as signal channels. During SystemC code generation, the modules from library are instantiated and connected to describe PEs and interconnection network according to the ADL description. Input and output ports of an array are automatically integrated in a standard interface module. At each cycle during simulation, this interface module reads values from a testbench file and assign them to respective input ports, and it writes out the data present at ouput ports. This interface allows the connection and co-simulation of other SystemC modules, such as the model of a host processor, through shared files. The C++ code, written manually to describe the functionality of FUs is also automatically integrated. The final result is a cycle accurate simulation model of the template. Additionally, this simulation model is still parameterizable, and parameter values may be altered through C++ `#define` directives.

Table 1 shows the size of four different templates written using our ADL and their corresponding generated code in SystemC (in lines of code without comments). The first example is a 2×2 fixed size array of PEs as described in [17]. The second example is a scalable array that uses the down-forward-hop network discussed in section 4.1. The third example is a template to execute a 4-point FFT. Its PEs vary in granularity and the interconnect network is highly specialized, which constitutes a good example for a heterogeneous array with irregular interconnect network. The fourth example is a weakly programmable processor array (WPPAs) template that was also independently described by Kupryanov[11] in MAML. The WPPA is an architecture that differentiate from ours in PE granularity and interconnect network. The WPPA PEs resemble a small instruction set processor and their network is implemented using wrapper/router modules.

Table 1: Lines of code for similar descriptions

	CRC-ADL	SystemC	Manual
2×2_Fibonacci	66	1890	1742
$w \times h$_NNDownFwd	62	1820	n.a.
8×6_4PointFFT	83	2010	n.a.
	CRC-ADL	SystemC	MAML
WPPAs_MAML	113	2315	550

The size of a description written using our ADL is on average 27 times smaller than the equivalent generated code in SystemC. When SystemC code is written manually, its size is only 8% smaller than the generated code and is still 25 times larger than the ADL description. That can be seen for the first example. We did not write manual code for the other examples because it is a very elaborate and time-consuming task. In the fourth example, we compare our ADL to MAML. We are able to describe the WPPA template in 113 lines of code, while its description in MAML took 550 lines of code[11]. These results show that (1)our ADL may be generically applied to describe CGRAs and (2) that we can write more concise description than up-to-date approaches.

5. CONCLUSIONS AND FUTURE WORK

We introduced one architecture description language designed to describe coarse-grained reconfigurable architecture templates. We presented key features and technical innovations to deal with specific challenges in the design phase of highly spatial architectures. The highlights of our ADL are: (1) we use a concise, definite and easy to understand non-XML based syntax, (2) we are able to describe intuitively the formation of irregular array structures, and (3) we introduce a new way to describe the interconnect network called *connection rules*. We discussed the potentialities of our ADL through two major examples: firstly, we demonstrate that it provides a natural way to formally validate the described templates. Secondly, we automatically generate a SystemC based simulator for the architecture template described with our ADL.

Up to now, our ADL does not support the description of pipelined or multi-cycle FUs. It is also not yet possible to integrate timing anotations. Most of these drawbacks are a consequence of using C++ to describe the functionality of FU modules. Therefore, our actual work concentrates on providing support to these features in our ADL.

The ADL proposed in this work allows template descriptions about 25 times smaller when compared to SystemC based descriptions. When compared to MAML ADL, we improve code size 4 times. Our ADL enables automatic tool generation, which facilitates the fast investigation of several architecture instances at the design phase. All these aspects suggests that the proposed architecture description language may boost the productivity at the design phase of coarse-grained reconfigurable arrays.

6. ACKNOWLEDGMENTS

This work is supported by Programme Alβan, European Union Programme of High Level Scholarships for Latin America, n. E04D045457BR and DFG Priority Program 1148 on Reconfigurable Computing Systems, RO1030/13-1 and RO 1030/13-2.

7. REFERENCES

[1] R. Azevedo, S. Rigo, and al. The ArchC architecture description language and tools. In *International Journal of Parallel Programming*, 2005.

[2] S. Bashford and al. The MIMOLA language version 4.1. Technical report, Lehrstuhl Informatik XII University of Dortmund, 1994.

[3] F. J. Bouwens, M. Berekovic, and al. Architectural exploration of the ADRES coarse-grained reconfigurable array. In *Proc. of Int. Workshop on Applied Reconfigurable Computing*, 2007.

[4] A. Fauth, J. V. Praet, and M. Freericks. Describing instruction set processors using nML. In *European Design and Test Conference*, 1995.

[5] D. Fischer, J. Teich, and al. Design space characterization for architecture/compiler co-exploration. In *Proc. of the International Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, 2001.

[6] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proc. of the Design Automation Conference*, 1997.

[7] A. Halambi and P. Grun. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. of the Design, Automation and Test in Europe*, 1999.

[8] A. Halambi, P. Grun, and al. Automatic software toolkit generation for embedded systems-on-chip. In *Proc. of the International Conference on VLSI and CAD*, 1999.

[9] R. Hartenstein, M. Herz, and al. Using the kressarray for reconfigurable computing. In *Configurable Computing: Technology and Applications*, 1998.

[10] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with LISA*. 2002.

[11] A. Kupriyanov, F. Hannig, and al. An architecture description language for massively parallel processor architectures. In *Proc. of the GIITGGMM*, 2006.

[12] B. Mei, S. Vernalde, and al. Adres: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Proc. of FPL*, 2003.

[13] B. Mei, S. Vernalde, and al. Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study. In *Proc. of the Design, Automation and Test in Europe*. IEEE Computer Society, 2004.

[14] T. Morimoto, K. Saito, and al. Advanced processor design using hardware description language aidl. In *Proc. of Asia and South Pacific Design Automation Conference*, 1997.

[15] N.Dutt and P. Mishra. Architecture description languages for programmable embedded systems. In *IEE Proc. of Computers and Digital Techniques*, 2005.

[16] T. Oppold, T.Schweizer, and al. CRC - concepts and evaluation of processor-like reconfigurable architectures. *it - Information Technology*, 2007.

[17] M. Rullmann, S. Siegel, and al. Efficient mapping and functional verification of parallel algorithms on a multi-context reconfigurable architecture. In *20th International Conf. on Architecture of Computing Systems - Workshop on Dynamically Reconfigurable Systems (DRS)*, 2007.

[18] H. Singh, G. Lu, and al. Morphosys: case study of a reconfigurable computing system targeting multimedia applications. In *Proc. of the Design Automation Conference*, 2000.

[19] T. Sugawara, K. Ide, and T. Sato. Dynamically reconfigurable processor implemented with IPFlex's DAPDNA technology. *IEICE Transactions on Information and Systems*, 2004.