Improving Hardware Reuse through XML-based Interface Encapsulation

Miguel A. Sánchez*, Marisa Lopez-Vallejo*, Carlos A. Iglesias† and Carlos A. Lopez-Barrio*
*Dpto. Ingeniería Electrónica †Dpto. Ingeniería de Sistemas Telemáticos
E.T.S.I. Telecomunicación
Universidad Politécnica de Madrid, Spain
*{masanchez, marisa, barrio}@die.upm.es †cif@gsi.dit.upm.es

***Abstract*—This work proposes an encapsulation scheme aimed at simplifying the reuse process of hardware cores. This hardware encapsulation approach has been conceived with a twofold objective. First, we look for the improvement of the reuse interface associated with the hardware core description. This is carried out in a first encapsulation level by improving the limited types and configuration options available in the conventional HDLs interface, and also providing information related to the implementation itself. Second, we have devised a more generic interface focused on describing the function avoiding details from a particular implementation, what corresponds to a second encapsulation level. This encapsulation allows the designer to define how to configure and use the design to implement a given functionality. The proposed encapsulation schemes help improving the amount of information that can be supplied with the design, and also allow to automate the process of searching, configuring and implementing diverse alternatives.**

***Index Terms*—hardware IP core; component interface; encapsulation; reuse;**

I. Introduction

Current technologies have allowed the integration of millions of transistors in a single chip. However, the complexity of hardware design makes the design cycle too long and complicated. A serious improvement in productivity is required, and this is only affordable by component reuse [1], which makes possible to cope with even larger designs, simultaneously reducing the time needed to get the final product available. Reused components, known as cores, may come from former designs or from third parties. In the last case, they are usually called by hardware designers *Intellectual Property (IP)*, because of licensing reasons.

System design is thus simplified by proper assembly of such cores, and the identification of the best ones and the parameters that better configure a given IP is typically a time consuming task. Hence, IP-based design actually needs tools to simplify component assembly and core generation, and also to provide quality assessment measures (area, speed, power, accuracy) that ease the selection. These goals can only be accomplished if the cores offer a good reuse interface to allow automated subcomponent customization, instantiation and interconnection, with the corresponding simplification of the design of hierarchical collections of modules.

Classical Hardware Description Languages (HDLs) like VHDL[2] or Verilog[3], are widely used to describe hardware cores. However, their interface offers limited possibilities for reuse. The part of the interface which comes with the core implementation is only used to declare the ports that connect the core with other components in a system, and parameters that allow configuring a limited set of design features.

In general, the remaining interface information, which is needed to integrate the core in other designs, is commonly appended as external documentation. This documentation usually includes parameter description and port functionality, as well as the relationships between these elements. VHDL/Verilog encapsulation is not able to consider this kind of information. Typical core documentation should include chronograms to describe component port protocols, functions that establish restrictions on parameter values, and even application notes to help the user of the core to understand how to use the design practically.

Furthermore, each core provides interface information in a different way. In the case of two cores implementing the same function, each interface is specifically designed to encapsulate a particular implementation, so it is strongly linked to implementation. To include a core described with traditional HDLs in a larger system often requires a deep knowledge of the implementation, which is a serious drawback if the automation of the design space exploration process is desired.

This work addresses the improvement of hardware reuse by defining two encapsulation schemes at different abstraction levels: structural and functional. Both schemes have been conceived with different goals to solve problems we found when trying to reuse IPs designed by means of conventional HDLs. The idea behind the proposed encapsulation is to extend the features of conventional HDL interfaces following the component-based design paradigm[4] to ease both the parameterization of designs and the reuse of previous IPs.

Even though VHDL or Verilog are the most extended HDLs for core descriptions, other languages have been proposed to describe reusable cores.

Previous proposals range from specific DSLs[5], for example HML[6] that targets hardware synthesizable descriptions, but also software languages have been used to describe hardware, such as C o Java, for instance Handel-C[7] or JHDL[8]. Although these DSLs were conceived to improve IP core description, their interface is still close to the one provided by conventional HDLs.

XML has also been used in the context of hardware de-

978-2-9541-8100-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICECCS.2012.21

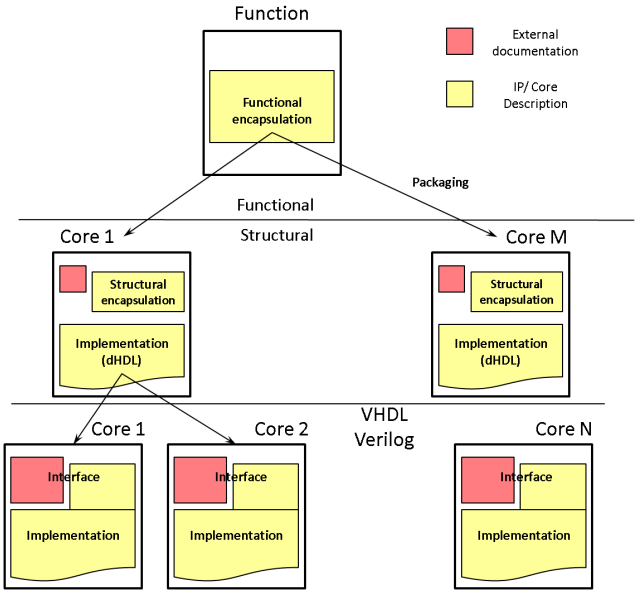

Fig. 1. Proposed encapsulation layers.

scription, for instance as intermediate language to carry out HDL code transformations[9] or to specify low level structural descriptions[10]. An approach closer to the work described here is[11], [12] where XML is used for interface specification to improve the reusability of cores in reconfigurable platforms. This approach uses XML to generate HDL wrappers for cores that are described in different languages or can even be generated from different tools. These wrappers are used within integration tools to implement more complex cores. The proposal is oriented to offer cores from different sources with a unified interface description, but higher abstraction levels are not considered. Also in this proposal it is not possible to access to implementation information because it is hidden.

Finally, IP customizable cores are available from commercial tools like Xilinx LogiCORE IP[13], whose limitations have in part inspired this work.

The structure of this paper is the following. First an overview of the proposed encapsulation layers will be presented. Then it will be described in detail the different proposed encapsulation schemes, structural and functional. A highly versatile component will be used as design example all through the paper. Finally some conclusions will be drawn.

II. Approach overview

A key goal of this work is to provide support to the utomatization of reuse-related tasks. These tasks cover the search, selection, configuration and implementation of previously designed hardware cores or IPs. Conventional HDLs were conceived for the description of hardware, being their interface just a list of input/output ports. Thus it is first necessary to improve the interface of conventional HDLs, to extend the information related to the implementation. This is carried out in the interface provided by the *structural encapsulation*. More complex configuration elements are defined enhancing their possibilities of configuration. Furthermore, it is possible to report on particular implementation data, what cannot be done in conventional HDLs. Finally, the proposed structural interface simplifies the automation of core interconnection and instantiation.

The structural encapsulation is specified with XML by using a template based on its definition in *XML-Schema*. This kind of specification is convenient to support tool development, in particular in our case we target the automation of IP selection, configuration and integration tools. It can be also used to implement automated design space exploration.

However, as its own name states, structural encapsulation is linked to a specific implementation. Elements declared in this encapsulation level are oriented to describe the underlying design, whereas this design is the implementation of a particular functionality. In the case of different cores implementing the same function we would find different interfaces. A second level of encapsulation specification is then required. The aim of this encapsulation is to describe the function without paying attention to specific implementation details. Instead, it will offer alternative implementations at a higher abstraction level, so it is called *functional encapsulation*.

Figure 1 illustrates the different abstraction levels in the proposed encapsulation scheme. From bottom to top we can find:

- At the lowest level there are conventional HDL cores. These cores provide a limited reuse interface. In order to complete the core interface specification, they must provide extensive external documentation, represented in the figure with the dark boxes.

- The next level corresponds to cores with interface based on structural encapsulation. More flexible than the VHDL interface, it also offers to the designer to include implementation information as part of encapsulation, so the amount of information that must be provided externally is reduced (dark boxes are smaller).
- At the upper level functional encapsulation is defined. This encapsulation is not tied to a particular implementation, but to a function description. It can cover possible implementations described by different structural encapsulates.

Also in Figure 1 the relationships between the proposed elements are shown:

- Structural encapsulation is linked to a core description with a domain specific language, dHDL [14]. This language has been designed to provide support to new characteristics defined into structural encapsulation. This language offers to the designer the possibility of wrapping HDL cores, but also a full design can be described with this language.
- Functional encapsulation does not have a specific implementation, however, multiple structural encapsulates can be configured to implement the same function. This is carried out through *packaging*.

Next, the proposed encapsulation schemes will be presented in detail.

III. Structural Encapsulation

As mentioned before, a first step to simplify the reuse of cores described with conventional HDLs is to improve the reuse interface which is attached to the implementation. Structural encapsulation is aimed at improving the possibilities for configuring the core, and incorporating information related to the implementation. The improvements offered by this encapsulation against the conventional approaches are:

- A core described with traditional HDLs typically uses parameterization as a method for reuse, where a single implementation can be adapted to different applications through micro-structural modifications. The parameters of these languages can only define a limited set of types and can only perform very simple function calls, so often structural modifications are limited to changes in bit-widths. Structural encapsulation enhances these possibilities, for example evaluating complex functions or even functions based on the value of other configuration elements.
- The structural encapsulation scheme allows to declare elements that inform about implementation characteristics (for instance, required area and latency, throughput, ...). This information, which is never provided by conventional HDLs, is useful to simplify automatic core integration in a new system, or even help in exploring the design space.
- To simplify component integration and interconnection, core ports are grouped into a single statement, which contains all HDL port declarations in a group, and it is oriented to implement a predefined input/output protocol.

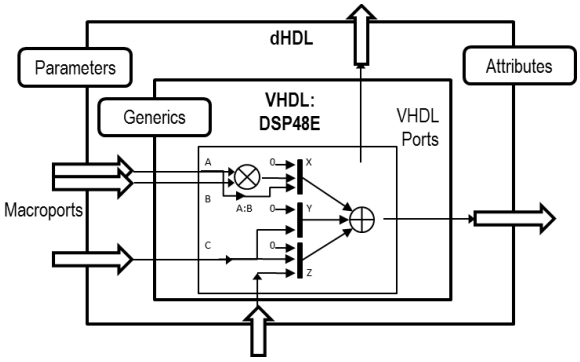

Fig. 2. Levels of encapsulation.

Therefore, the structural encapsulation declares as configuration elements the `parameters`, as reporting elements the `attributes`, and `macroports` as elements which define the input/output.

Wrapping one or more VHDL cores using structural encapsulation is the simplest mechanism which allows to use the proposed elements and improve their reuse. Figure 2 represents the correspondence between elements defined in the structural encapsulation and elements defined in a VHDL interface (ports and generics). We have used as example for component encapsulation, a well-known low level element of current Xilinx FPGAs, the DSP Slice [15]. DSP Slices are designed to implement most of the basic operations related to signal processing (mainly based on additions and multiplications). They are extremely useful, but their interface is rather complex both regarding the number of ports and configuration options. We target several objectives when defining the structural encapsulation of this component:

- Information only provided by data-sheets regarding configuration parameters, ports, and the relationships among them is incorporated to the interface. This way, the instantiation of the desired component is simplified.
- Constraints for the configuration space can be carried out by avoiding misconfigurations.
- Most of unused ports and parameters can be hidden.
- Significant attributes can be incorporated, reporting information like the latency or throughput that the component provides.

Figure 3 shows the structural encapsulate which corresponds to the wrapping for the component in Figure 2.

Structural encapsulation is defined in four sections:

- `Definitions`, which provide a space to define restricted types, based on unions and ranges, from basic types (int8, int16, float, etc.).
- `Parameters`, where component configuration elements can be declared. Both basic or user defined types can be used for the declaration of parameters.

```
<declaration coreid="dhdl_dsp48e"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="xsdhdl.xsd">
  <definitions>
    <deftype id="optype">
      <union type="string">
        <item>MULT</item>
        ...
      </union>
    </deftype>
  </definitions>
  <parameters>
    <parameter id="AREG">
      <type>boolean</type>
      <value>true</value>
    </parameter>
    ...
  </parameters>
  <attributes>
    <attribute id="LATENCY">
      <type>int</type>
      <value>3</value>
    </attribute>
    ...
  </attributes>
  <macroports>
    <macroport id="DSP_A">
      <type>lvi</type>
      <property id="width">
        <value><vall>25</vall></value>
      </property>
    </macroport>
    <macroport id="DSP_B">
      <type>lvi</type>
      <property id="width">
        <value><vall>18</vall></value>
      </property>
    </macroport>
    ...
</declaration>
```

Fig. 3. Structural encapsulation: DSP Slice from Xilinx.

- `Attributes` allow to declare and initialize elements which will be used to report on component characteristics.
- `Macroports` are related to input/output declarations.

A. Definitions section

A set of basic data types are the basis for declaring the elements of next sections: `string`, `boolean`, `int8`, `int16`, `int32`, `float`, `double`. This section allows defining new types based on the basics, by defining ranges and unions. Figure 4 shows an example of range declaration.

```
...
<deftype id="a_port_widthtype">
  <range type="int8">
    <from>2</from>
    <to>25</to>
    <step>1</step>
  </range>
</deftype>
...
```

Fig. 4. Range Declaration: constrained input type.

The DSP Slice component does not allow an arbitrary configuration value for its ports bit width. There are multiple ports defined in the component, for both data and control. For instance, port `A` and `B` are used to provide input data and their implementation is fixed to 25 and 18 bits respectively. Figure 4 shows how to define a range which will be used to restrict configuration options related to the component port `A` up to 25 bits.

B. Parameters section

Parameters are the elements of the structural encapsulation that define the configuration options of the implementation. They are similar to VHDL generics, but their definition is significantly improved. First, parameters can be constrained using the previously defined types. Example of Figure 5 shows how to declare the parameter that will configure `A` port width using the type predefined in Figure 4.

```
...
<parameter id="a_data_width">
  <type>a_port_widthtype</type>
  <value><vall>25</vall></value>
</parameter>
...
```

Fig. 5. Parameter Declaration.

Parameters can be also initialized by means of the value returned from a function. Functions admit as arguments other parameters, so it is possible to establish relationships between parameters. Back to the DSP Slice example, certain modes of operation are only available if other options have been previously configured. For instance, structural encapsulation allows defining a function that can check if a given requirement to set a mode is met.

C. Attributes section

If the parameters can be considered as an input to the design configuration, attributes can be seen as output elements for reporting information from the implementation. Attributes provide a simple mechanism to report characteristics of a design, hiding implementation details.

```
...
<attribute id="latency">
  <type>int8</type>
  <value>4</value>
</attribute>
...
```

Fig. 6. Attribute declaration.

Attribute values are obtained after implementing the design, so they can be used to inform on design characteristics. For example, Figure 6 shows the declaration of attribute `latency` for the DSP example. It will return the latency in cycles required to perform an operation. This attribute depends on how configuration is translated to implementation, and it can vary from 0 to 4 cycles.

D. Macroports section

Macroports define the set of signals that are input and output to the design. In contrast to conventional HDL port declaration, a macroport groups a whole set of signals in a single declaration. Furthermore, the macroport declaration establishes the functionality and timing characteristics that the set of signals

must meet, that is the protocol. This element of the structural encapsulation simplifies the process of interconnecting and synchronizing signals between different cores, and therefore helps when building tools that support reuse.

```
...
<macroport id="DSP_A">
  <type>lvi</type>
  <property id="width">
    <value><val>"a_data_width"</val></value>
  </property>
</macroport>
...
```

Fig. 7. Macroport declaration.

Figure 7 shows an example with the macroport declaration for `A` port in the DSP Slice component. This is a data port so it is declared with a type compatible with the macroport definition, `lvi`, which is an input data vector synchronized with enable and clock signals. The chosen type allows the configuration of some parameters, like the `width` of the vector, which are configured by means of previously declared parameters.

IV. Functional Encapsulation

Structural encapsulation provides a high degree of parameterization, but it is also possible to incorporate further information in the core interface. This information can be used to automatically integrate the component within another design.

In this case, in addition to information related to the implementation itself, it is interesting if the interface incorporates information defining how a core can be used. For this purpose a new encapsulation level has been added. Two elements compose this new level. On the one hand, the functional encapsulation itself, which is oriented to describe a function independently from how it is implemented. On the other hand, a *packaging* of the structural encapsulation, which connects both encapsulation levels. With the packaging the designer defines how to configure the component to implement the function described by the functional encapsulate.

Functional encapsulation does not declare parameters that configure a particular implementation. The functionality is described by a set of `features` that the design must fulfill. Therefore, functional encapsulation is not a substitute for the structural encapsulation, but it is based on this. Actually functional encapsulation offers a feature-based description of the functionality that must be implemented by a design, but it is the designer responsibility to transform these features into a valid configuration for the structural encapsulate of a core.

Functional encapsulation is divided into three sections:

- `Function declaration`, which provides an identifier and information of the described function.
- `Features`, which allow to refine the function declaration.
- `Flows`, which model input/output to/from the declared function.

```
...
<function id="mult">
  <description>...</description>
  <library>arith</library>
  <version>
    <ver n="1">
       <description>...</description>
    </ver>
  </version>
</function>
<features>
  <feature id="precision">
    <description>...</description>
    <version ver="1">
      <description>...</description>
      <union type="string">
        <item>byte</item>
        ...
        <item>long</item>
      </union>
    </version>
  </feature>
  ...
</features>
...
```

Fig. 8. Functional encapsulation: multiplication.

A. Function declaration

The most important information related to the functional encapsulation is the function itself. A function is declared with an identifier within a namespace. The identifier must be unique in the namespace and descriptive enough for the function. This identifier will be used as a reference for designs that implement it. Figure 8 begins with `mult` function declaration within namespace `arith`.

The function section also describes version fields which allow to introduce modifications in the encapsulation, while maintaining the compatibility with already defined implementations from structural encapsulation.

B. Features

Function description can be refined by declaring features. Features have been designed to cover different objectives:

- Classify implementations. For example, a feature can be defined to separate between implementations with fixed or floating point arithmetic.
- Parameter configuration. For instance, a feature may define the bitwidth of the function.
- Design space exploration. For example, a feature can be defined to bound the area or performance of the implementation.

Figure 8 shows the declaration of a feature that defines the `precision` for the function `mult` within a set defined by a union. Structural encapsulation can use this feature to configure input bit-widths, or to configure if a given implementation can use rounding/truncation options. It can be used to determine if a design meets the minimum required precision.

C. Flows

Input-output data to/from the function is modeled by flows. Flows are a convenient mechanism for describing data streams. A flow collects the basic input-output information: number and type of each piece of data, and even basic timing.

```
...
 <flow id="input_op" io="in">
   <description>Input data stream.</description>
   <version ver="1">
     <description>...</description>
     <dt>
       <mean>1.0</mean>
       <peak>0.0</peak>
     </dt>
     <op>
       <opid id="op_a" type="int8"/>
       <opid id="op_b" type="int8"/>
     </op>
   </version>
 </flow>
 ...
```

Fig. 9. Flow specification.

As can be seen in Figure 9, a flow declaration defines:

- Normalized data rates that characterize the performance and timing information associated to the set of input/output data.
- A list of the different operands or input/output data subjected by the flow.

Whereas macroports from the structural encapsulation define a specific signaling required by the implementation, the objective of flows is to provide a mechanism to uniformly describe different implementations.

V. Packaging

Packaging is the last element of the proposal, and establishes the link between both encapsulations.

- It is an extension of structural encapsulation which allows the designer to define how an IP or design implements a function described by a functional encapsulate
- It is used to translate features which have been restricted, into valid configurations and interconnections for the design.
- It allows to verify if the implementation obtained from a configuration satisfies the constraints imposed by the features.

Packaging is divided into three sections. The first section, `function`, establishes the link between structural and functional encapsulations. Then a `configuration` section is devoted to specify how to configure parameters or macroports through feature and flow definitions and values. Finally a last section, `verification`, checks if all the functional features are fulfilled.

Figure 10 shows an example of packaging for the DSP Slice component previously introduced. This packaging maps the structural encapsulate of the DSP Slice with a functional encapsulation of `mult`, expressing that this component can perform the function `mult`.

VI. Exploration of Alternatives

The main purpose of defining the functional encapsulation is to support the exploration of alternatives when designing complex systems. This encapsulation level provides the access point to select and configure among the possible available implementations for a function. Both encapsulation levels

```
<function>
  <core id="dhdl_dsp48e" />
  <proto library="arith"
      id="mult" ver="1" />
</function>
<configuration>
  <parameter id="AREG">
    <value><vall>true</vall></value>
  </parameter>
  ...
  <parameter id="OPERATION">
    <value><vall>MULT</vall></value>
  </parameter>
</configurtation>
<verification>
  <feature id="precision">
    <value default="false">
      ...
    </value>
  </feature>
</verification>
...
```

Fig. 10. Packaging DSP48E: Multiplication.

can interact in a descending and ascending flow as will be described next.

A. Finding alternatives

Figures 11 and 12 show the relation between the elements proposed when exploring alternatives. Encapsulates and packagings are stored into different libraries.

Packaging allows to define the correspondence between functional and structural encapsulates. It specifies how to configure the structural encapsulation from the elements defined in the functional encapsulation (top-down flow), and also checks if the configuration obtained satisfies the constraints defined at the functional level (bottom-up flow).

To obtain an implementation from a functional encapsulate corresponds to a broad process, where it is unknown which is the implementation that better fits the requirements. This process is designated as finding alternatives. The objective of this process is to offer to the designer, among the possible implementations represented by the packaging, if a design or set of designs are close to fulfill the requirements defined by the function.

The search for alternatives process is therefore divided into two processes:

1) The first process seeks for the implementation of a VHDL core based on information supplied from the functional encapsulation. This process is represented in Figure 11. In this top-down process, where a user (the system designer) configures the elements defined in the functional encapsulation (features and flows), generating a set of constraints that must be met by all candidate implementations. Then those packagings which offer implementing that function translate the constraints into the parameters defined in the interface constituted by their structural encapsulation. Then dHDL components configured through structural encapsulation are generated and produce the corresponding VHDL core.

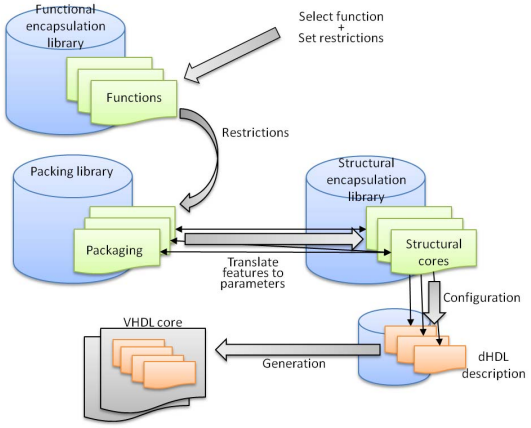

Fig. 11. Top-down process: finding alternatives.

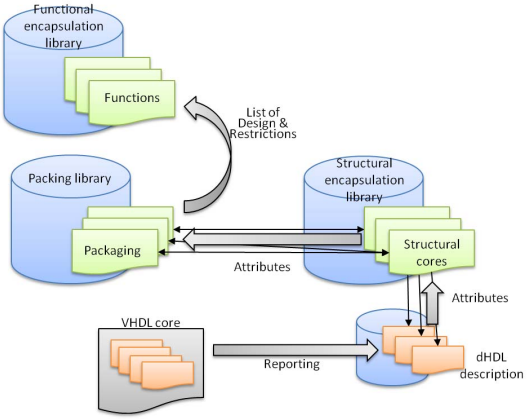

Fig. 12. Bottom-up process: validate implementations.

2) The second process, bottom-up, is shown in Figure 12. Once several alternatives have been generated, the verification process starts. The previous process generates VHDL cores, but also the information related to the code generation process, which is collected by the structural encapsulate through attributes. Then, with the returned information the packaging verification starts, which is oriented to determine what functional features are met by the given implementation. At the functional encapsulation level, the user obtains a list of possible implementations of the function, and also how many constraints have been satisfied by each design.

B. Example

Figure 13 shows an example of the application of the proposed reuse methodology to encapsulate a previously designed component, the well-known DSP Slice.

1) The structural encapsulate and dHDL language define a wrap for the VHDL component with an extended reuse interface.
   - The new encapsulate offers the same set of parameters and ports for communication than the VHDL component instance.
   - Structural parameters allow to verify the correctness of the values which will be used to configure the hardware core. It is also possible to define relationships between parameters.
   - Depending on the chosen configuration, it is possible to enable and disable ports though the dHDL code.
   - Design attributes have been defined to inform about the particular implementation under study; their value is calculated when the VHDL code is generated.
2) The component can implement multiple functions, such as addition or multiplication. Packaging is used to define how to configure the structural encapsulation to implement each function (top-down flow).
   - Packaging allows the designer to specify how to configure and how to connect the component to implement the function.
   - Packaging also defines how to check from the information obtained through structural attributes, if they really meet the constraints imposed by the function.
   - Packaging allows to offer the design as a solution to implement the function.
3) At functional level, the DSP Slice packagings are alternatives of implementation. Other components may also offer an implementation with another different packaging.
   - Functional encapsulation does not describe specific implementations, but features that may have multiple implementations.
   - Once features of the functional encapsulation are set, they become constraints that must be verified by the candidate implementations.

VII. Conclusion

This paper has presented a hardware reuse approach based on the specification of two encapsulation layers: structural and functional encapsulation. Structural encapsulation is close to the implementation, but improves the information that can be offered by designs described using conventional HDLS. This encapsulation layer defines parameters, attributes and macroports as key elements of the reuse interface of a hardware core. Functional encapsulation is used as a higher abstraction level to describe a functionality by means of features and flows.

The proposal adds new elements to improve the interface information of the cores described in conventional HDLs in order to support the creation of reuse tools. For instance, the proposed encapsulation can simplify the process of exploring alternatives when integrating a new core or IP in a complex design.

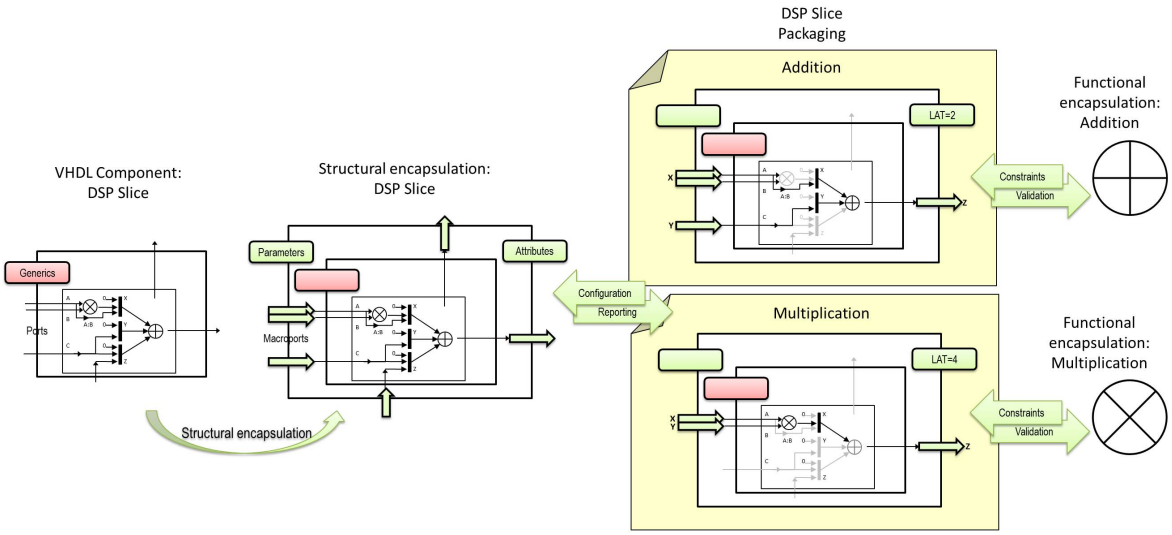

Fig. 13. The DSP Slice example illustrates all proposed encapsulation levels.

ACKNOWLEDGMENT

This work was funded by the CICYT project DELTA TEC2009-08589 of the Spanish Ministry of Economy and Competitiveness and Avanza Telcodev TSI-020100-2010-1092 of the Spanish Ministry of Industry, Tourism and Commerce.

REFERENCES

[1] D. Gajski, "IP-based design methodology," in *Proceedings. 36th of the Design Automation Conference, 1999.* IEEE, 1999, p. 43.

[2] D. L. Perry, "VHDL : Programming by Example," *New York*, p. 476, 2002.

[3] J. Bhasker, *A Verilog HDL Primer*. Star Galaxy Press, 1997.

[4] E. A. Lee and A. L. Sangiovanni-Vincentelli, "Component-based design for the future," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2011, pp. 1–5.

[5] A. Raja and D. Lakshmanan, "Domain Specific Languages," *International Journal of Computer Applications IJCA*, vol. 1, no. 21, pp. 105–111, 2010.

[6] M. Leeser, "HML, a novel hardware description language and its translation to VHDL," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 1, pp. 1–8, Feb. 2000.

[7] M. Aubury, J. Saul, G. Randall, and R. Watts, "Handel-C Language Reference," 1996.

[8] M. J. Wirthlin and B. McMurtrey, "IP delivery for FPGAs using applets and JHDL," in *Proceedings of the 39th annual Design Automation Conference*. ACM, 2002, pp. 2–7.

[9] J.-H. Oetjens, R. Görgen, J. Gerlach, and W. Nebel, "An automated flow for integrating hardware IP into the automotive systems engineering process," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 1196–1201.

[10] C. Fulong, Z. Zhaoxia, and F. Xiaoya, "XML component-based modeling for digital circuits," in *2010 Second Pacific-Asia Conference on Circuits, Communications and System*. IEEE, Aug. 2010, pp. 148–151.

[11] N. Rollins, A. Arnesen, and M. Wirthlin, "An XML Schema for Representing Reusable IP Cores for Reconfigurable Computing," in *2008 IEEE National Aerospace and Electronics Conference*. IEEE, July 2008, pp. 190–197.

[12] A. Arnesen, N. Rollins, and M. Wirthlin, "A multi-layered XML schema and design tool for reusing and integrating FPGA IP," in *2009 International Conference on Field Programmable Logic and Applications*. IEEE, Aug. 2009, pp. 472–475.

[13] Xilinx, "Xilinx LogiCORE IP, http://www.xilinx.com."

[14] M. A. Sánchez, M. López-Vallejo, and C. A. Iglesisas, "xHDL: Extending VHDL to improve core parameterization and reuse," *IEEE International Symposium on Parallel and Distributed Processing with Applications, (ISPA-12)*, July 2012.

[15] Xilinx, "Virtex-5 FPGA XtremeDSP Design Considerations User Guide, http://www.xilinx.com."