Chapter 7

DESIGN AUTOMATION

Integrating TLM in SoC Design Flow

Christophe Amerijckx[1], Stephane Guenot[2], Amine Kerkeni[3], Serge Hustin[1]

STMicroelectronics Belgium[1]; STMicroelectronics France[2]; STMicroelectronics Tunisia[3]

Abstract: Although the TLM development and usage only require a C++ development environment and a SystemC library, design automation is the key to integrating TLM in the SoC design flow for further reaping the design productivity and quality rewards brought by TLM. This chapter explains how TLM has been integrated in the design flow at STMicroelectronics both by extending the SPIRIT XML packaging standard to support TLM and by developing the tools needed to integrate TLM in the flow.

Key words: SPIRIT; SystemC TLM; SoC; design flow; design automation; XML; data model and schema; platform assembly; meta-level; content-level; configurator; generator; netlister; IP packaging; platform generation.

1. INTRODUCTION

The minimum tool and library requirements for the TLM methodology are simply a C++ development environment and SystemC classes. A flawless integration of the TLM methodology into the SoC design flow, however, entails further tools and libraries implementations. This chapter describes at length the necessary accompanying implementations to make the best use of the TLM methodology in the SoC design cycle.

This goal is attainable through establishing and enforcing a standard *automation* strategy to integrate the TLM methodology into the essential phases of the SoC design flow. The TLM assembly should therefore adopt an automation approach ranging from the design database to editor, configurator, checker, and netlister.

F. Ghenassia (ed.), Transaction Level Modeling with SystemC, 241-266.

From a given design database, TLM components are instantiated, configured, and interconnected by a platform editor. Configurators and checkers are subsequently employed to propagate the redundant information through the design description, and to verify the design integrity. Lastly, netlisters project the design description into its different targets, covering not only TLM but also verification, software, and hardware emulation.

To support all of the automation tools above, a *common format* must be adopted to store the design information as well as to package the design components.

The remainder of the chapter provides a detailed description of the component and design representations, followed by an in-depth explanation of the automation tools, and finally an illustration of their applications on a real design.

2. DESCRIPTION OF DESIGN AUTOMATION

2.1 Introduction

With the advent of the explosive nanotechnology era, the design of the System-on-Chip (SoC) is getting increasingly complex without the help of efficient tools. The SPIRIT[1] Consortium [1] has developed a standard mechanism for describing and handling IPs, with the aim of accelerating large-scale SoC designs through automated configuration and integration of the designs [2].

SPIRIT provides an eXtensible Markup Language (XML) schema to describe components and designs. Rules and regulations are also imposed by SPIRIT for implementing the user interfaces of automation tools, such as generators and configurators, to handle SPIRIT compliant components or designs. The SPIRIT design environment is clearly illustrated in Figure 7-1.

For each component or IP in a given SoC design, SPIRIT defines a specific XML file containing the metadata that will be used by a SPIRIT compliant design tool. The content of such component XML file is defined in the SPIRIT schema for the following aspects:

1. *Bus interfaces available for a component*.
   This description allows the automation of connecting a component to different components of the same interfaces. The bus interface here refers

[1] Structure for Packaging, Integrating and Re-using IP within Tool-flows.

to a bus definition that specifies the bus Vendor Library Name Version (VLNV).

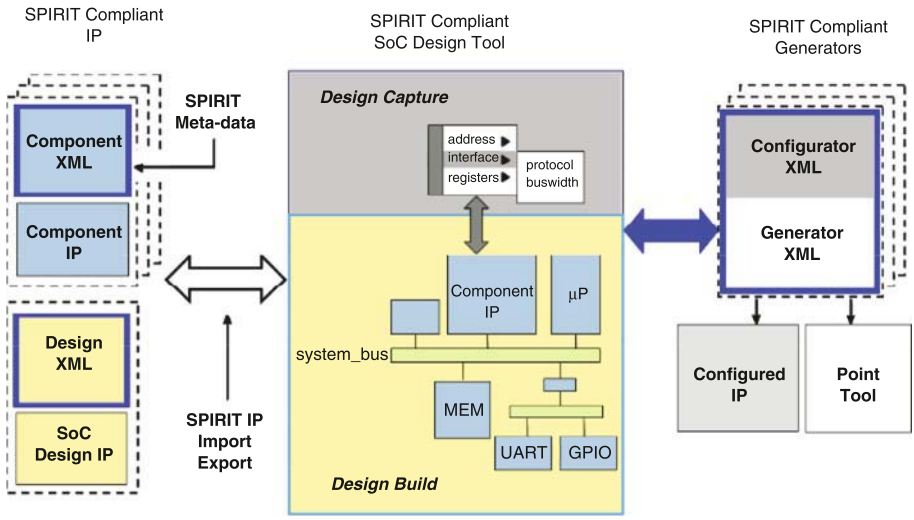

Figure 7-1. SPIRIT Design Environment

2. *Different views available for a component.*
   The descriptions of the different views available for a component are essential in determining the abstraction levels for that component within a given design. Each view refers to a file set that holds all of the files specific to that view.
3. *Memory map and remap information.*
   Providing the memory map and remap information of a component is intended for specifying the different registers available on the slave interface of that component.
4. *Address space.*
   The address space of a component must be described because it defines the logical space accessible by the master interface of the component.
5. *Hierarchy information.*
   If a component is hierarchical, the information of the different associated component instances must be provided along with their interconnections.

6. *Configuration parameters*.
   The different parameters available for configuring a component are described, for example, the size of a RAM, the number of master/slave of a bus, etc.
7. *File sets of different component views*.
   A list of the file sets that specifies the various files used by each view of a component has to be provided.

A SPIRIT compliant tool uses the content of the SPIRIT metadata to automate the SoC design through the followings:
1. instantiation of components in a given design followed by prompting users for the view selection;
2. automatic connection of components depending on their bus interfaces;
3. prompting users for the configuration of parameters;
4. launching SPIRIT compliant generators.

The top-level structure of a SoC design is specified by its different component instances along with their connection at the bus interface level via interconnections, or at the point-to-point level via ad hoc connections for all the signals not belonging to a bus. The ad hoc connections are of course present only at the RTL level.

A SPIRIT generator can be launched from a design tool to accomplish those tasks that are not managed by the design tool, for instance, netlist generation, configuration, compliancy, consistency checking, clock-tree generation, etc.

The coming sections further describe how the design automation is achieved in line with SPIRIT. The initial version, i.e. SPIRIT V1.0, focuses only on the RTL hardware view. We shall therefore start our discussion from this single-view approach. The next major release, i.e. SPIRIT V2.0, will allow the co-existence of the multiple-view such as untimed TLM, timed TLM, bus-cycle accurate (BCA), and RTL. Our discussion will highlight the ST Microelectronics view on how such multiple-view approach is completed.

2.2 Single-View Component Structure

The single-view (i.e. RTL) component structure pertaining to the SPIRIT V1.0 standard is detailed hereafter.

2.2.1 Bus Definition

The specification of a bus is stated by a bus definition. The bus definition is identified by its VLNV. Another important piece of information stated

within the bus definition is the maximum number of masters and slaves that a bus can hold. In addition, the bus definition contains information specific to the RTL view. This includes the collection of signals that belong to the bus, and of the constraints to be applied to these signals such as directions on master/slave and default values.

Indeed, the bus definition is strongly analogous to VHDL in the sense that the bus definition is a kind of VHDL record type whereby a bundle of signals belonging to the same group can be defined.

2.2.2 Bus Interface

Different components are interconnected by the bus interfaces defined for each component. Each bus interface within a component is designated a specific name. Two interfaces can be connected together if:

1. they are of the same type as identified by the VLNV of the bus definition;
2. they have matching interface natures (e.g. master, slave).

For this reason, a bus interface must be specified in terms of its type and nature as illustrated by the following example.

Example of Bus Interface Definition

```
<spirit:busInterface>
    <spirit:name>ambaAPB</spirit:name>
    <spirit:busType spirit:vendor="AMBA" spirit:library="AMBA"
spirit:name="APB"
      spirit:version="v1.0"/>
    <spirit:slave>
      <spirit:memoryMapRef spirit:memoryMapRef="ambaAPB"/>
    </spirit:slave>
    <spirit:signalMap>
      <spirit:signalName spirit:busSignal="PSELx">psel</spirit:signalName>
      <spirit:signalName spirit:busSignal="PENABLE">penable</spirit:signalName>
      <spirit:signalName spirit:busSignal="PADDR">paddr</spirit:signalName>
      <spirit:signalName spirit:busSignal="PWRITE">pwrite</spirit:signalName>
      <spirit:signalName spirit:busSignal="PWDATA">pwdata</spirit:signalName>
      <spirit:signalName spirit:busSignal="PRDATA">prdata</spirit:signalName>
    </spirit:signalMap>
</spirit:busInterface>
```

At the RTL level, an interface represents a group of signals. The signal names defined in the bus definition may not always match the signal names defined in the design. Therefore, a section of *signal mapping* is included in the bus interface for indicating the relationship between these two signals. As shown in the example above, the signals defined in the bus definition (i.e.

logical signal names) are mapped to those defined in the design (i.e. physical signal names).

Similar to the analogy of the bus definition with VHDL, the definition of a bus interface looks like a signal definition whose type would be the one defined for the bus definition using a record.

2.2.3 View Specification

In SPIRIT, the *view* of a component represents a level of abstraction or an implementation of a particular component. The specification of a single-level view is quite straightforward. First of all, a given view must be designated a specific name in order to distinguish it from other views. An environment identifier further specifies the environment that can be used by the given view, for instance, simulation and synthesis environments. The language applicable in the given view such as VHDL, Verilog or SystemC must be stated as well. Lastly, a reference to a file set listing all the files delivered with that particular view is specified, along with the HDL-specific model names. Quoted below is an example of the specification for a component with the RTL view.

Example of the RTL View Specification

```
<spirit:view>
   <spirit:name>RTL</spirit:name>
   <spirit:envIdentifier>Simulation</spirit:envIdentifier>
   <spirit:envIdentifier>Synthesis</spirit:envIdentifier>
   <spirit:language>vhdl</spirit:language>
   <spirit:modelName>leon2_Uart(struct)</spirit:modelName>
   <spirit:fileSetRef>vhdlSource</spirit:fileSetRef>
</spirit:view>
```

The SPIRIT V1.0 standard assumes that each view contains all of the defined bus interfaces, implying that bus interfaces are neither optional nor specific to a particular view.

2.3 Multiple-View Component Structure

The multiple-view (i.e. untimed/timed TLM, BCA, and RTL) component structure proposed by ST Microelectronics for the SPIRIT V2.0 standard is detailed hereafter.

2.3.1 Bus Interface

To support multiple hardware views, SPIRIT V2.0 will add a new element in the bus interface schema to specify the name of the available

abstraction level for the bus interface. The specification of signals is shifted to the abstraction level of RTL. As an example, a given bus interface may support all of the abstraction levels available for a design while another bus interface may only support RTL and TLM abstractions.

One may claim that such added feature is duplicated information because the abstraction level of the bus interface should be the same as the one of the component. However, a specific abstraction level of a given component may not always have the same name as the corresponding abstraction level in the bus definition. The naming convention helps to enforce a mapping of the abstraction levels between the component and the bus definition. For example, the abstraction of the timed transaction level could be named as "TLM-timed" in the bus definition while it could be named as "PVT" in the component.

2.3.2 View Specification

The view specification of a component with multiple abstraction levels remains the same as the view specification of the single-view component. The only difference is that meticulous care must be given to handle bus interfaces. In multiple-view components, certain bus interfaces may be present in a particular view but absent in another view. A typical example of such is the test interface that exists at the RTL but not at the untimed TLM, as depicted in Figure 7-2. Note that this a distinct difference from the SPIRIT V1.0 where all bus interfaces are defined for every component view.

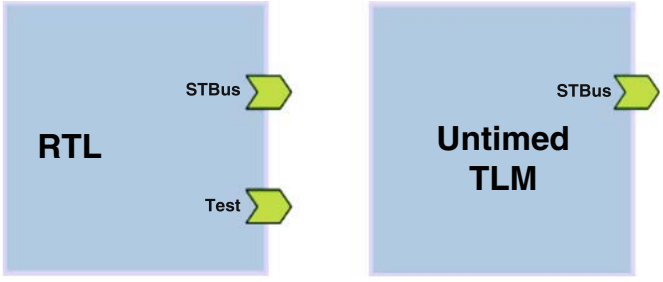

Figure 7-2. Bus Interfaces of a Component with Two Abstraction Levels

To resolve such problems, each component view must include the list of bus interfaces available in a design. This feature facilitates the SPIRIT compliant tool to retrieve the available bus interfaces for each abstraction level of the component.

Consider the example of a component with two abstraction levels, RTL and untimed TLM, as illustrated in Figure 7-2. At the RTL level, the component has two bus interfaces. The first is a STBus level-2 interface while the second is a test interface. At the untimed TLM level, a STBus level-2 interface is the only present interface. The corresponding XML code of this multiple-view component is provided hereafter. Note that *busInterfaceNameRef* refers to the name of the bus interfaces defined in the bus interface section of the component XML metadata file.

Example of the XML Code for a Multiple-View Component

```
<spirit:views>
    <spirit:view>
        <spirit:name>RTL</spirit:name>
        …
        <spirit:interfaceList>
            <spirit:busInterfaceRef>
                <spirit:name>STBus<spirit:name>
                <spirit:busAbstraction>RTL</spirit:busAbstraction>
            </spirit:busInterfaceRef>
            <spirit:busInterfaceRef>
                <spirit:name>test<spirit:name>
                <spirit:busAbstraction>RTL</spirit:busAbstraction>
            </spirit:busInterfaceNameRef>
        </spirit:interfaceList>
    </spirit:view>
    <spirit:view>
        <spirit:name>PV</spirit:name>
        …
        <spirit:interfaceList>
            <spirit:busInterfaceRef>
                <spirit:name>STBus<spirit:name>
                <spirit:busAbstraction>PV</spirit:busAbstraction>
            </spirit:busInterfaceRef>
        </spirit:interfaceList>
    </spirit:view>
</spirit:views>
```

2.4 Design Structure

In our context, a *design* is the representation of the top-level structure for a given SoC platform. A SPIRIT compliant design contains all of the instances and connections that form a SoC.

There are three compulsory sections of a SPIRIT compliant design:

1. VLNV of the top-level design;
2. different component instances instantiated at the top-level;

3. connections between component instances at the bus interface level and point-to-point level.

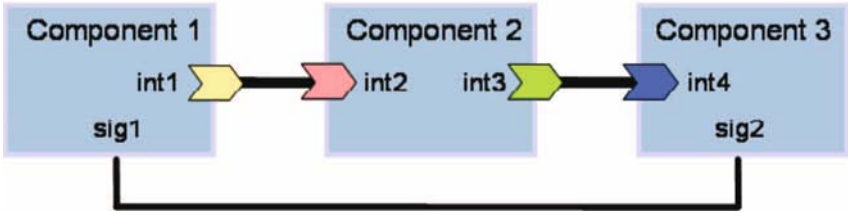

Figure 7-3. A Design with Three Components

Figure 7-3 demonstrates a simple design made of three components interconnected by:

a) bus interfaces from int1 to int2, and from int3 to int4;
b) point-to-point connection between sig1 and sig2.

The corresponding XML description of this component is provided hereafter. The description starts with the VLNV of the design, giving information on the design vendor, library, name, and version. The next section, <*spirit: componentInstances*>, lists all of the component instances. Each of these instances holds an instance name and a reference to access the component library identified by its VLNV. Following this is the section of <*spirit: interconnections*>, which provides the information on the interconnections of bus interfaces. For every component involved in the interconnection, there are two values required by this section: the name of the component instance and the name of the bus interface on that component. The last section, <*spirit: adHocConnections*>, gives the specification of the point-to-point connections between signals.

Example of XML Design Representation

```
<spirit:design>
   <spirit:vendor>ST</spirit:vendor>
   <spirit:library>Example</spirit:library>
   <spirit:name>simple_design<spirit:name>
   <spirit:version>1.0</spirit:design>
   <spirit:componentInstances>
      <spirit:componentInstance>
         <spirit:name>component1</spirit:name>
         <spirit:componentRef spirit:vendor="ST" spirit:library="Processor"
            spirit:name="proc1"  spirit:version="1.0"/>
      </spirit:componentInstance>
      <spirit:componentInstance>
```

```
        <spirit:name>component2</spirit:name>
        <spirit:componentRef spirit:vendor="ST" spirit:library="Bus"
          spirit:name="bus1"  spirit:version="1.0"/>
      </spirit:componentInstance>
      <spirit:componentInstance>
        <spirit:name>component3</spirit:name>
        <spirit:componentRef spirit:vendor="ST" spirit:library="Peripherals"
          spirit:name="uart" spirit:version="1.0"/>
      </spirit:componentInstance>
    </spirit:componentInstances>
    <spirit:interconnections>
      <spirit:interconnection
        spirit:component1Ref="component1" spirit:busInterface1Ref="int1"
        spirit:component2Ref="component2" spirit:busInterface2Ref="int2"/>
      <spirit:interconnection
        spirit:component1Ref="component2" spirit:busInterface1Ref="int3"
        spirit:component2Ref="component3" spirit:busInterface2Ref="int4"/>
    </spirit:interconnections>
    <spirit:adHocConnections>
      <spirit:adHocConnection>
        <spirit:pinReference componentRef="component1" spirit:signalRef="sig1"/>
        <spirit:pinReference componentRef="component3" spirit:signalRef="sig2"/>
      <spirit:adHocConnection>
    </spirit:adHocConnections>
</spirit:design>
```

No information regarding the abstraction level of the design components is provided in the XML description. This aspect will be handled in a separate file. One way to handle this is providing a default file that gives the preferred list of view for each component instance. The tool will check through the default list for the *first* available view. Once found, that view will be accepted by the tool. The default rule can be overwritten by another if the designer would like to change the selected view. This is quite a common routine in the design process where all of the components are initially at untimed TLM level, and then some components are gradually changed into RTL. Such manipulation continues until a complete RTL platform is obtained at the end.

Since no information is given for the abstraction level in the design, the role of generators is vital. The netlister must verify the abstraction level for each component to assure the right connections. Two component interfaces of the same abstraction level can obviously be connected directly. If their abstraction levels are different, the netlister will have to insert a transactor - or BFM - between the two interfaces. This is typically the case where a component is of untimed TLM level while another is of RTL level. In such cases, the role of the transactor will be converting transactions into signals.

3. AUTOMATION TOOLS

3.1 The Need for Platform Assembly Automation

The Integrated Circuits (IC) industry has been growing exponentially for a few decades. ICs are no more simple chips with a few components for a specific functionality but System-on-Chip (SoC) with multi-millions gates for a whole system. The current SoC industry must treat every step in the SoC design flow as much as possible at the platform level, where the system behavior is studied and managed through the communication between platform IPs.

Once a given SoC design is simulated at TLM level, it will have to go for the RTL simulation where all cycle-accurate signals must be connected. Not only is the RTL simulation a time- and effort-consuming job, it is also highly error-prone.

Consequently, the need for the platform assembly automation has become more and more critical nowadays. The concept of using the data model in the platform assembly operation is relatively clear for its users. However, this concept is not really implemented in the industry today because the optimization of the high-level platform simulation does not allow treating such data models (which represent very often the transactions between platform components).

With the ever-rising SoC complexity, the need for automating the SoC platform assembly should no longer be compensated by such optimization. The most noticeable advantages include reduced error probability and immediate productivity enhancement. In addition, any modifications on an existing platform description can save SoC developers a lot of time. The SoC flow automation needs to tackle two areas:

1. automation of standard tasks by using SPIRIT compliant generators;
2. providing SPIRIT compliant tools for tasks that require user inputs.

3.2 Foundation of Flow Automation

All kinds of industrial activities follow specific procedures that can be described as a *flow*. Without exception, the SoC industry must obey this rule as well. Various methodologies were developed to assist SoC developers in formalizing the SoC design flow. Formalizing the SoC design flow means identifying tasks that can be fully automated and those that require user input.

Note that the majority of the automated tasks can only be executed upon the input of the necessary information from users. To create such formalization, two parts of the flow must be distinguished:

1. *Structural Part*. This part consists of user-dependent data, environment-dependent data, configuration parameters, data constraints, and any other data that are collected as input for a set of automation tools.
2. *Functional Part*. The functional part is the data treatment in a flow.

Both parts can be formalized into the format of Unified Modeling Language (UML), which is a methodology describing the flow of any activities.

Since long, there exists an absolute formalized data model in the algebra called Relational Data Base (RDB). This universal data model is quite simple but very efficient. Through a data model diagram, the RDB represents the formal structure and logical relation of the input data for a given activity flow. Otherwise stated, the RDB describes and outlines the structure of data.

Figure 7-4 illustrates an example of the data model with its data classes, data fields, and the data hierarchy. This diagram is an interpretation of the SPIRIT data model that covers only the description of hardware connections. It shows the non-exhaustive examples of data classes and data fields, and cardinal relations between the classes.

There are loads of different methods to realize the diagram or *schema* of the RDB model. To design this schema, the data structure that will be described by the RDB model must be carefully developed, including the hierarchical and cardinal relation for the whole data structure.

Many commercial data model tools called database engines are available in the market today. These tools access to the database through the standard request language, Simple Query Language (SQL). However, these tools often require proprietary database engines to store data in the proprietary binary format. These proprietary tools have consequently made SQL a non-standard language without a universal format.

For this reason, a descriptive language should be used to handle the structural part of a flow. The XML is strongly recommended as the best solution for this purpose since it can hold any formalized contents. The XML documents can be manipulated by using standard parsers and validators such as Xerces from W3C[2]. These parsers and validators check the consistency of the described elements in the XML (like what an RDB engine

[2] Refer to World Wide Web Consortium (W3C) at http://www.w3.org.

will do), e.g. checking the descriptions of platform design, components/IPs, and bus specifications/definitions.

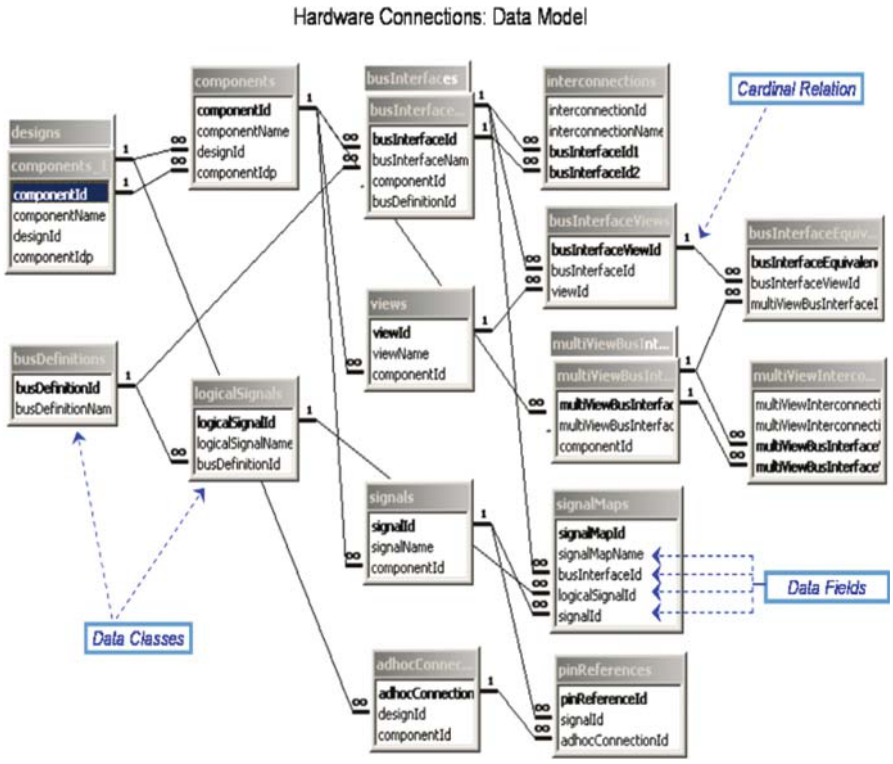

Figure 7-4. Example of Data Model

Although the XML can handle data contents very well, the data structure must be described. Different methodologies are available to provide the technical support for describing data structures. Be it any method, a list of "containers" has to be designated to describe the data model. These containers are the data classes depicted in Figure 7-4; they are called *table* in the RDB, *class* in the object-oriented language, and *sequence* in the XML Schema Definition[3] (XSD). For each container, a set of data fields must be defined as depicted in Figure 7-4. These data fields are called *column* in the RDB, *member* in the object-oriented language, and *element* in the XSD.

[3] Recommended by W3C to formally describe the elements in the XML documents.

The real strength of a data structure descriptive tool lies in its ability to illustrate the relation between the container elements, rather than the description of the container itself. There are two fundamental types of such relations distinguished by their cardinality:

1. *One-to-One Relation*. A given data member can only be related to a single data member from another data class, for example, a set of parameters added to a given class.
2. *One-to-Many Relation*. A given data member can be related to one or more data members from other data classes, for example, a microelectronics component can hold several bus interfaces.

The data model exists most of the time in an implicit form in many industrial activities. The explicit implementation of the data model, on the other hand, is able to offer very extensive applications ranging from assembly tools to generator tools.

SPIRIT has adopted the XML format, a universal standard opened to the public, to describe a data model. As such, the XML specification cannot verify the relational and cardinal integrity of the data model described by the XML documents. A list of semantics rules called "grammar" or "schema" must be implemented to further describe the data model. Certain specific languages can formalize the XML semantics rules; among which, the most advanced description methodology is the XSD. The XSD schema is however limited to describe all the necessary semantics rules. For this reason, there are two principal parts in the SPIRIT standard:

1. XSD schema for describing the technical content;
2. User guide for describing the semantics rules and the design automation flow.

3.3 Strategy for Automation Tool Development

This section discusses the strategy adopted for developing the automation tool in line with the SPIRIT standard.

3.3.1 SPIRIT Meta-Level Description

Be it any methodology or description language of data model, the structure of the data model, i.e. the data classes, data fields, and the relation between data classes, must be described by formal rules. These rules are *meta-level* rules since they implement the "data model" of a data model, i.e. *meta* data model.

All of the description methodologies provide the syntax to create and update the meta-level description. In the SPIRIT standard, the meta-level

descriptions for XML documents are implemented in XSD schema. The semantics rules in charge of validating the meta-level description itself are usually hard-coded in the database engine, for instance, hard-coded in Xerces for the XSD schema.

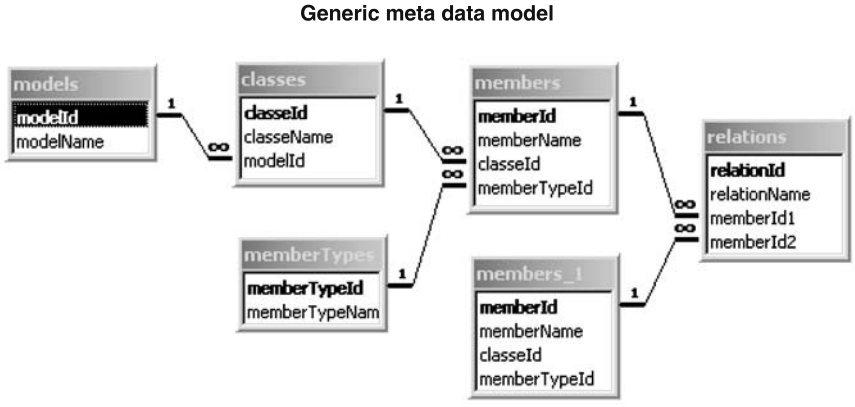

Figure 7-5. Description of Generic Meta Data Model

Figure 7-5 gives a description of the generic data model at meta-level. Note that in a SPIRIT meta data model, a list of data classes (also known as table or sequence) must be defined. Every defined data class holds a list of members (also known as element or column) with a unique type each. The cardinal relation between the members can be described as well.

3.3.2 SPIRIT Content-Level Description

Once the SPIRIT meta data model is implemented, its structure must be filled up with the necessary contents[4]. The SPIRIT XML documents are created to store the database of such contents.

An important goal of the SPIRIT platform description is to provide an input for a set of tools such as generators. Various Application-Programming Interfaces (API) can be used to implement the SPIRIT compliant tools. For any kinds of API as C++ or Java, the data must be treated first. To do so, the API must contain a list of methods to create, modify, and delete the SPIRIT objects such as bus interfaces.

[4] See section 3.4.2 about the editor tools for editing the contents.

The W3C consortium provides a specific API, Xerces, which is divided into two main parts:

1. *Validator and Parser*: parse XML documents and then load the XSD schema; also check the content integrity of XML documents, and fill a proprietary Xerces structure if the integrity is well respected.
2. *Document Object Model (DOM)*: a low-level API.

The DOM API provides very generic structures in the form of a simple data tree. It also provides methods to manage the data tree, i.e. methods to access the SPIRIT contents through the software. However, the DOM API does not take into account the XSD schema and thus not allowing efficient programming of SPIRIT compliant tools. A higher layer dedicated to the SPIRIT schema is therefore required as explained in the next section.

3.3.3 API Generation

As explained earlier, a higher layer API is necessary to handle the SPIRIT schema or the meta data model. A standard approach to produce this layer is writing it manually in an appropriate programming language such as C++ or Java. This manual task is always a very time- and effort-consuming job due to the huge schema size.

A more interesting approach is to make use of the SPIRIT schema to generate the SPIRIT-specific API structures, and their *exhaustive* access methods to the entire content described by the XSD schema. This approach is feasible because the meta-level description is formalized in the XSD schema, which is indeed a kind of XML documents. The standard DOM API allows loading the XSD schema for filling up the corresponding meta data models in the form of C++ meta structures. Essentially, such C++ structures are filled in the memory after being analyzed by the model builder tool as the representation of the SPIRIT meta data model.

Once this abstract representation is available, the different applications can be generated, e.g. all the translation tools from/to the SPIRIT XML format. The most important generation is the SPIRIT API source code, which represents a “snapshot” of a given SPIRIT version. This allows an automatic re-generation of API source code if the SPIRIT version is updated or when proprietary schema extensions are implemented. The choice of the target language for the generated API is independent of the language of the generator itself; thus, it can be in any language such as C++ or Java[5].

[5] For the future version of SPIRIT platforms, such generations are in C++ for simulation compliance reasons.

With this generated API, SoC developers will no longer have to deal with a generic data tree structure based upon DOM but concrete SPIRIT compliant objects such as components and bus interfaces. All of the elements of the SPIRIT schema such as signal direction or signal size will be created explicitly during the API generation as the class members. These data members can be manipulated explicitly by the SPIRIT developer.

The dedicated C++ structures with access methods to the members are not the only necessary targets. The generated API must provide the methods to instantiate, update, duplicate, and remove SPIRIT objects. However, a further need is an immediate method to load the XML document into the C++ structure, and to dump this structure in a new XML document after modification. This is where a loader and a dumper are required. Figure 7-6 and 7-7 summarize the discussion of the previous three sub-sections.

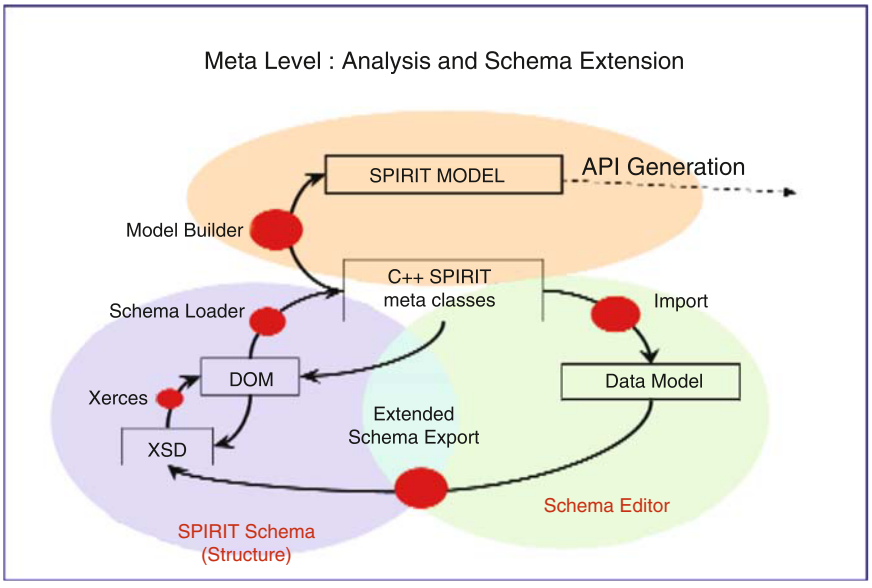

Figure 7-6. SPIRIT Meta-Level

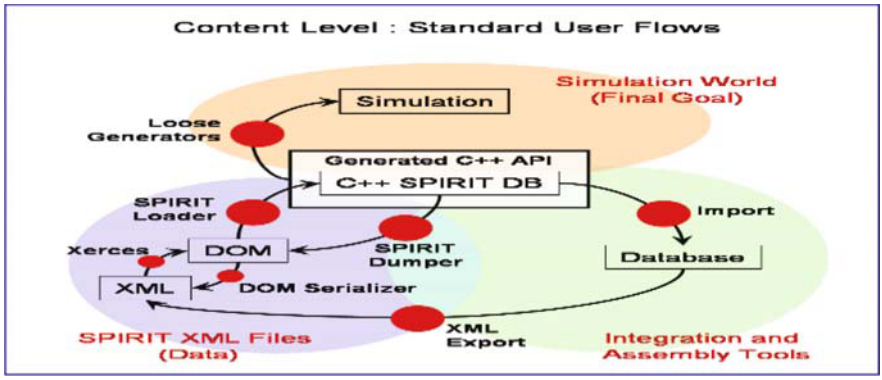

Figure 7-7. SPIRIT Content-Level

3.3.4 Development Environment and Inter-operability

The SPIRIT development environment is the "meta-level environment" for the SoC design environment, which is indeed analogous to the relationship between meta-level and content-level SPIRIT data models. The development environment provides SoC developers with the necessary tools and facilities to create a user-specific design environment.

To optimize the work of SPIRIT developers, a full SPIRIT development environment including the generated API should be made available. The generated API is nevertheless an exhaustive C++ view of SPIRIT schema without the semantics rules. As mentioned earlier, the SPIRIT semantics rules are collected in the SPIRIT user guide. To enable verifying these rules throughout the development, a higher API layer must be implemented manually[6].

This higher API facilitates the writing rules by encapsulating the low-level methods to complement the work of the generated API. Furthermore, this layer separates the generated API from the tools developed by SPIRIT developers, i.e. SPIRIT tools are independent of the generated API. As the SPIRIT version evolves, tool developers simply need to re-generate the API for the XSD schema and update the API for the semantics rules while the SPIRIT tools themselves are kept untouched.

Another fundamental goal of SPIRIT is to provide a tangible solution of flexible inter-operability among different tools provided by different EDA vendors or IP providers. SPIRIT describes the generators of the SoC design flow with standard interfaces at every flow step. The SPIRIT development

[6] STMicroelectronics has developed a checker tool to verify the SPIRIT semantics rules (see section 3.4.4).

environment contains various "building bricks" such as Loose Generator Interface (LGI) and Loose Generator Change (LGC) (see section 3.4.5), which allow end users to swap from one SPIRIT compliant tool to another at any step throughout the flow.

3.4 SPIRIT Compliant Automation Tools

At the entry point of the SPIRIT design flow, the data required in the flow activities must first be imported either manually or automatically by some scripts or tools. Once entered in the SPIRIT environment, the imported data will be typed or edited by specific editors. The SPIRIT compliant automation tools will then treat the data in line with the objective of end users, which is the SoC design simulation for SoC developers.

The SPIRIT compliant automation tools are classified into five families:

1. Packager.
2. Editor.
3. Checker.
4. Configurator.
5. Generator.

3.4.1 Packager

Automated processes require the input data to be packaged or put together according to the technical specification. The SPIRIT compliant *packager* is an automation tool for packing all the input data into the XML description of microelectronics components.

If the input data exists already in a specific formal format such as FrameMaker, RTL or SystemC, a set of scripts are provided to translate such data into a SPIRIT XML document.

3.4.2 Editor

Once the imported data is translated into the XML format by the SPIRIT compliant packager, the design description requires some meta data that is necessary to enable the design configuration and automation. Such meta data is usually prepared and entered manually by SoC developers. For this reason, an editor is needed in the flow to edit and modify the SPIRIT XML database.

As an ASCI format, the XML document allows any text editors to process and package a component description. The end users, however, expect something more user-friendly than the XML format such as an editor with a Graphical User Interface (GUI).

Two types of GUI editors are available:

1. *XML Generic Editor*. An editor that edits and modifies the XML documents based on any XSD grammar, e.g. SPY.
2. *SPIRIT-Specific Editor*. A specific editor that respects the SPIRIT grammar and schema. The XSD methodology adopted by SPIRIT is consistent enough to provide automatic ways to create GUI for XML packaging. This is a tool generation that is similar to the API generation from the same representation of the SPIRIT meta data model (see section 3.3.3). The most important packaging parts, however, remain the manual optimization process.

The result of an editor is an XML document for all of the components of a design. A particular tool is needed to assemble all of these components and interconnect them to form a design or platform. Therefore, a specific editor, *platform assembler*, is created to perform this job.

The platform assembler can be in the form of GUI where users can select any components to instantiate in the platform, interconnect bus interfaces, and connect signals not belonging to any bus by ad hoc connections. In addition, the assembler tool also configures the parameters of the component instances. The result of the platform assembler is a new XML document file for the design, i.e. Design XML file.

3.4.3 Checker

The SPIRIT compliant checker is a specific tool developed for verifying the semantics rules written in the SPIRIT user guide.

The SPIRIT schema allows checking many integrity constraints in a design description. However, certain description methodologies cannot check all of the constraints needed to verify a design description.

Therefore, an applicative layer called *checker* is added on Xerces, the standard XML validator. This layer implements the semantics rules for the SPIRIT schema and the reference validity for elements from different files.

The latter task cannot be performed by the XSD since it can only treat a single file at a time. As an example, the checker must be used to verify the names of the components instantiated in a design because all of the components have their own separate XML documents. The checker can be invoked anytime at any step in the design flow.

3.4.4 Configurator

The SPIRIT compliant configurator is a tool that configures the SPIRIT data according to the design context. This configuration is based on either:

1. a template, e.g. configuration for replicating interfaces;

2. or the user input in line with the design context.

Just like the checker, the configurator can be invoked can be called repeatedly after any iteration from the Design XML.

Indeed, the configurator is an XML-to-XML tool. Given the SPIRIT input as an XML document, it is configured by the SPIRIT configurator to produce a SPIRIT output that is another XML document but with some modifications or configurations.

3.4.5 Generator

The SPIRIT compliant generator is a very important tool family. In the design flow, several generators can appear together as a *generator chain* with each targeting a specific task. For the SoC design flow, the key generator is of course the netlister (see section 3.5 for further discussions).

Typically, a generator reads a complete SPIRIT design description as the input data in the XML format. A generator chain is then created for that design. Each generator of the generator chain, for the reason of inter-operability, contains three elements of SPIRIT compliant generators:

1. *Loose Generator Interface (LGI)*.
   This sub-generator takes the design environment as the input to create an LGI file, which holds the access path to all of the XML documents of the design environment. It helps to implement a generator tool that is independent of the design environment, i.e. only the LGI will have to be modified if the access path to an XML file is changed.
2. *Function-Specific Generator*.
   This is a generator with specific tasks, e.g. the netlister to produce a netlist.
3. *Loose Generator Change (LGC)*.
   If any function-specific sub-generators make some changes in XML documents, then the LGC must write these changes into an LGC file to update the design environment.

Indeed, the generator chain is described in the XML format as a "meta-generator" that permits the SPIRIT compliant tools to perform the entire generation flow from the design description in a single shot. The meta-generator generates all of the necessary output of the design (e.g. netlist), which will be utilized by the simulation tool such as the SystemC-RTL simulator.

3.5 Netlister

The netlister is a particular type of generator tool that plays a vital role in the SPIRIT SoC automation flow. Recall that the main objective of the

SPIRIT standard is to automate the SoC design flow from the XML design description to an operational simulation or implementation. The netlister can therefore be considered as the most important automation tool.

The standard input for the SoC simulation are formal sources as RTL or SystemC, which can be compiled and executed by a simulation kernel. Be it any level of abstraction, a top netlist is required to perform the platform simulation, e.g. an RTL/TLM top netlist is necessary for simulating a mixed RTL/TLM design. The netlist is a purely structural description that is essentially a list of component instances with interconnections between their interfaces. Note that no algorithmic or behavioral codes should be included in a netlist.

For any given platform, either an RTL or SystemC netlist is required for its simulation. Thus, two types of netlisters are available in general:

1. RTL netlister to generate the RTL netlist;
2. SystemC netlister to generate the SystemC netlist (for TLM).

3.5.1 Co-Simulation Netlist

The co-simulation is typically a simulation that mixes both TLM and RTL models. It is needed for simulating a complex RTL design with a huge number of elements to support high-level functionalities. Recall that this is indeed the reason of developing the TLM methodology to represent the behavior of an RTL design with only the algorithm using a system-level language.

The users will construct a mixed test-bench by choosing the appropriate IPs to be simulated at RTL as the Design Under Test (DUT), and those to be simulated at TLM for its behavior from the system point of view. Once the choices are made, the netlister tool will provide the corresponding netlist automatically.

3.5.2 Co-Emulation Netlist

To increase the execution speed of a simulation, certain synthesizable RTL blocks can be mapped on emulators. A different netlist than the co-simulation netlist must be generated by the netlister tool.

3.6 Other Generators

Many other kinds of generators can be created according to what the users need to do. The only condition to create a generator is that the necessary information must be written in the XML document.

3.6.1 Regression Generator

Many of the SoC platform IPs especially the host processor need to execute the embedded software. The role and the amount of the software has become increasingly significant in the SoC design. Therefore, it is very useful to describe a list of test codes to execute an IP for a full regression test. Such description can be written in an XML document, based on which the regression test suite can be generated.

3.6.2 Register Access Test Generator

In the SPIRIT schema, the memory map of an IP can be described with the accurate descriptions of all of its registers and register fields. Specific generators can generate the SystemC header files for the IP, which contains the definitions of all registers. These generators can also generate the software that will be executed on the simulation platform. This software will try to access the IP registers to verify if the access rules are well respected.

4. EXAMPLE

4.1 Platform Architecture

The TC4SoC[7] platform serves as a demonstrator to validate the design automation strategy described earlier on a real platform. TC4SoC is a test chip vehicle that validates several IP blocks and CAD tools in 90nm CMOS technology. It is a SoC design comprising PCI and LMI interfaces. It provides the STBus External Port (SEP) that enables the interconnection to external high-speed buses. This chip supports flash memories and a board range of peripherals connected via a programmable glue logic. Other memories included are embedded ROM, SRAM, and eDRAM. IPs are interconnected through the STBus interconnect, which contains four STBus nodes as transaction routers. Figure 7-8 illustrates the structure of TC4SoC.

[7] A SoC design developed by STMicroelectronics.

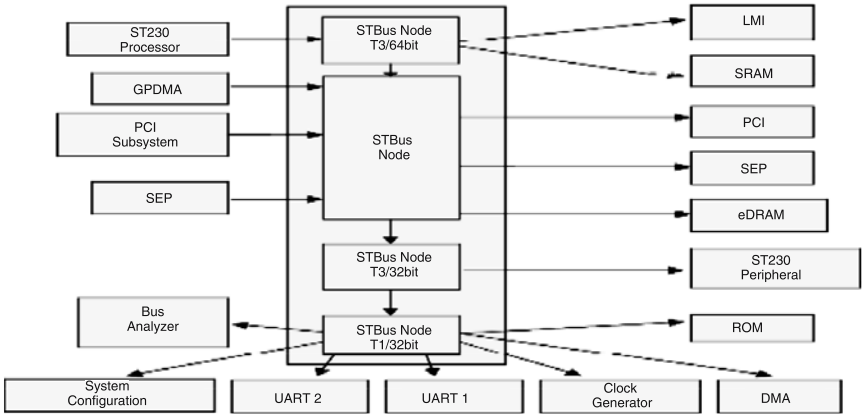

Figure 7-8. The Architecture of TC4SoC Platform

4.2 IP Packaging and Platform Generation110

This section describes the IP packaging and the platform generation following the SPIRIT strategy. The UART IP in the TC4SoC platform will be given as an example of the SPIRIT compliant component.

To begin with, a SPIRIT component file must be created for the UART. The RTL entity (VHDL in this example) of the UART is used as the entry point to create this component file. The signal section under the *hwModel* section of the component file corresponds to the signal list of the RTL entity. This mapping can be done manually or automatically by a tool, *vhdl2spirit*.

If signals are correctly named in the RTL entity, for instance, giving the same prefix to a group of signals belonging to the same bus interface and using the standard names to indicate bus types, then the *vhdl2spirit* tool is able to detect bus interface type and create the *busInterface* section with the corresponding signal mapping.

As depicted in Figure 7-9, the example of UART component has three bus interfaces, i.e. a slave STBus T1 interface, an input clock interface, and an input reset interface. There are three types of STBus signals: T1, T2, and T3. Since the STBus interface used in the UART is only T1, all of the STBus signals are prefixed with *stbus1* in the UART VHDL entity. As a result, the *vhdl2spirit* tool can detect correctly the STBus interface. The script can also detect automatically that the interface is T1 since T2/T3 signals are missing.

After instantiating all the design IPs, users are now ready to connect them to the STBus interconnect. The STBus interconnect is a standard yet fully configurable IP. For instance, the number of bus interfaces is not static as it depends on the number of IPs connected to it. It is therefore impractical to store all of the possible interconnections in a database.

The XML database contains a template of the STBus interconnect. This template is processed by a style sheet in the eXtensible Stylesheet Language (XSL) to generate an XML file with the appropriate number of bus interfaces.

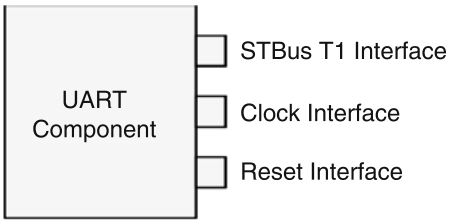

Figure 7-9. Bus Interfaces of UART in TC4SoC Platform

The remaining tasks include the configuration of signal size and the pin connection between master and slave interfaces, which are performed by a SPIRIT generator. First, the SPIRIT design environment generates a Loose Generator Interface (LGI) file that describes the environment. This file is indeed the input file to help the generator to locate the paths to different XML files.

When the design and all component instances are loaded, the configuration generator will search for the STBus interconnect instances. Once the interconnect instances are identified, the configuration generator will loop through the STBus interfaces of these instances. For each interface, the generator will try to match its signal sizes to the connecting interface with respect to the STBus specification. Once matched, the interfaces are interconnected. When the interconnection task is completed, the generator will check and remove any unused signals in the STBus interfaces. A new XML file is then generated with the information of the new path for the interconnections. This file is passed to the netlister in the form of the Loose Generator Change (LGC) file.

Tasks accomplished up to this point include the platform configuration, the IP configuration, and the connections of the design-level bus interfaces. The netlister tool can now perform its job with all the available information to generate a top-level RTL netlist of the TC4SoC platform for simulation. Figure 7-10 shows the SPIRIT design automation flow.

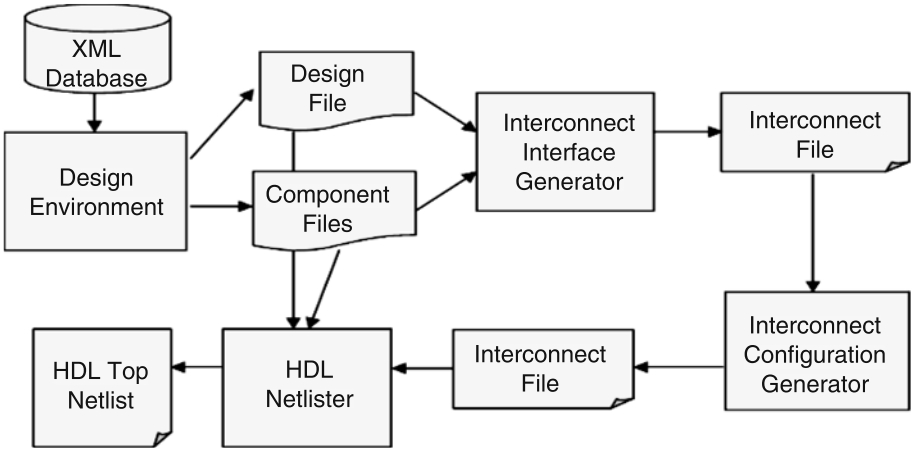

Figure 7-10. SPIRIT Design Automation Flow

REFERENCES

[1] SPIRIT Consortium web site available at: http://www.spiritconsortium.org

[2] SPIRIT Schema Working Group, “Spirit User Guide V1.0”, December 2004.