Draft Standard for the IP-XACT meta-data and tool interfaces

Prepared by the

Schema Working Group
of
The SPIRIT Consortium

Abstract: The *IP-XACT Standard forms the conformance checks for XML data designed to describe electronic systems. The meta data forms which are standardized include: components, systems, bus interfaces and connections, abstractions of those buses, and details of the components including address maps, register and field descriptions, and file set descriptions for use in automating design, verification, documentation, and use flows for electronic systems. The standard includes a set of XML schemas of the form described by the World Wide Web Consortium (W3C) and a set of semantic consistency rules (SCRs). The standard also provides for a generator interface that is portable across tool environments. The specified combination of methodology-independent metadata and the tool-independent mechanism for accessing that data provides for portability of design data, design methodologies and environment implementations.*
Keywords: *Electronic Design Automation, EDA, XML Design Meta Data, IP-XACT, XML Schema, Tight Generator Interface, TGI, Semantic Consistency Rules, SRCs, Design Environment, Use Models, Tool And Data Interoperability, Implementation Constraints, Register Transfer Logic, RTL, Electronic System Level, ESL, Bus Definitions, Abstraction Definitions, and Address Space Specification.*

The SPIRIT Consortium.
1370 Trancas Street #184, Napa, CA 94558

Introduction

This introduction is not part of the Draft Standard for the IP-XACT meta-data and tool interfaces.

The purpose of this standard is to provide the electronic design automation (EDA), semiconductor, electronic intellectual property (IP) provider, and system design communities with a well-defined and unified specification for the meta-data which represents the components and designs within an electronic system. The goal of this specification is to enable delivery of compatible IP descriptions from multiple IP vendors; better enable importing and exporting complex IP bundles to, from and between EDA tools for SoC design (system on a chip design environments); better express configurable IP by using IP meta-data; and better enable provision of EDA vendor-neutral IP creation and configuration scripts (*generators*). The data and data access specification is designed to coexist and enhance the hardware description languages (HDLs) presently used by designers while providing capabilities lacking in those languages.

The SPIRIT Consortium is a consortium of electronic system, IP provider, semiconductor, and EDA companies. IP-XACT enables a productivity boost in design, transfer, validation, documentation, and use of electronic IP and covers components, designs, interfaces, and details thereof. It is extensible in specified locations.

IP-XACT enables the use of a unified structure for the meta specification of a design, the components that is based on manual or automatic methodologies. IP-XACT specifies the tight generator interface (TGI) for access to the data in a vendor-independent manner.

This standardization project provides electronic design engineers with a well-defined standard that meets their requirements in structured design and validation and enables a step function increase in their productivity. This standardization project will also provide the EDA industry with a standard to which they can adhere and which they can support in order to deliver their solutions in this area.

The SPIRIT Consortium has prepared a set of bus and abstraction definitions for several common buses. It is expected, over time, that those standards groups and manufacturers who define buses will include IP-XACT XML bus and abstraction definitions in their set of deliverable. Until that time, and to cover existing useful buses, a set of bus and abstraction definitions for common buses has been created.

A set of reference bus and abstraction definitions allows many vendors who define IP using these buses to easily interconnect IP together. The SPIRIT Consortium posts these for use by its members, with no warranty of suitability, but in the hope that these will be useful. The SPIRIT Consortium will, from time-to-time, update these files and if a Standards body wishes to take over the work of definition, will transfer that work to that body.

These reference bus and abstraction definition templates (with comments and examples) are available from the public area of the http://www.spiritconsortium.org web site.

Notice to users

Errata

Errata, if any, for this and all other standards of The SPIRIT Consortium can be accessed at the following URL: http://www.spiritconsortium.org/releases/errata/. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations, users guides, examples, etc. can be accessed at the following URL: http://www.spir-itconsortium.org/tech/docs/.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith.

Participants

The following members and observers took part in the IP-XACT Schema Working Group (SWG) and the Electronic System Level (ESL) Working Group (EWG):

Greg Ehmann, NXP Semiconductors, *Chair SWG*
Jean-Michel Fernandez, Cadence, *Chair EWG*
Gary Delp, LSI Corporation, *Technical Director*
Joe Daniels, *Technical Editor*

ARM: Allan Cochrane, Christopher Lennard, Andrew Nightingale, Chulho Shin, Peter Grun, Anthony Berent, Sheldon Woodhouse

Cadence: Jean-Michel Fernandez, Giles Hall, Saverio Fazzari, Victor Berman

CoWare: Cesar A. Quiroz, Kris Dekeyser

Denali: Gary Lippert

Infineon: Wolfgang Ecker, Thomas Steininger

LSI: Gary Delp, Wayne Nation, Gary Lippert, Dave Fechser

MatiTech: Aaron Baranoff

Mentor: John Wilson, Gary Dare, Mark Glasser, Matthew Ballance, Mike Andrews, Ajay Kumar

NXP Semiconductor: Geoff Mole, Ahmed Hemani, Roger Witlox, Greg Ehmann, Maurizio Vitale, Erwin de Kock

Prosilog/Magillem: Stephane Guntz, Cyril Spasevski

Sonics: Kamil Synek

ST Microelectronics: Christophe Amerijckx, Serge Hustin, Anthony McIsaac, Stephane Guenot

Synopsys: Mark Noll, Bernard DeLay, John A. Swanson, Paul Wyborny

Texas Instruments: Bob T. Maaraoui, Bertrand Blanc

Special acknowledgment is given to:

Mentor: Contribution of initial schema upon which the work is based

Synopsys: Contribution of constraint structure

The Board of Directors of The SPIRIT Consortium active during the release of the IP-XACT Standard:

Ralph vonVignau, NXP, *President*
Christopher Lennard, ARM, *Vice-President*
Lynn Horobin, *Executive Secretary*

John Goodenough, ARM
Stan Krolikoski, Cadence
Luke Smithwick, Kathy Werner, Freescale
Jean Bou-Farhat, Gary Delp, LSI
Bill Chown, Mentor Graphics
Bart de Loore, NXP Semiconductors
Serge Hustin, ST Microelectronics
Pierre Bricaud, Synopsys
Loic Le-Toumelin, Texas Instruments

Contents

Draft Standard for the IP-XACT meta-data and tool interfaces

1. Overview

This clause explains the scope and purpose of this standard; gives an overview of the basic concepts, major semantic components, and conventions used in this standard; and summarizes its contents.

1.1 Scope

This standard describes an eXtensible Markup Language (XML) data format and structure, documented with a schema[1] for capturing the meta-data which documents design intellectual property (IP) used in the development, implementation, and verification of electronic systems. The standard also includes a tight generator interface (TGI) to provide consistent, tool-independent access to the meta-data. The XML documents described and validated by the schema comprise a standard method to document IP that is compatible with automated integration techniques. The TGI provides a standard method for linking generation tools into a system development framework, enabling a more flexible, optimized development environment. Tools compliant with this standard shall be able to interpret, configure, integrate, and manipulate IP blocks that comply with the proposed IP meta-data description. This standard is independent of any specific design process. It also does not cover the behavioral characteristics of the IP.

1.2 Purpose

This standard provides a well-defined XML schema for meta-data that documents the characteristics of IP required for the automation of the configuration and integration of IP blocks; and also defines a TGI to make this meta-data directly accessible to automation tools.

1.3 IP-XACT design environment

While the document formats are the core of this standard, describing the IP-XACT specification in the context of its basic use-model, the design environment (DE) more readily depicts the extent and limitations of the semantic intent of the data. The DE coordinates a set of tools and IP, or expressions of that IP (e.g., models), through the creation and maintenance of a meta-data description of the SoC such that its system-design and implementation flows are efficiently enabled and re-use centric.

[1] IP-XACT uses the World Wide Web Consortium (W3C) standard for the eXtensible Markup Language (XML) data. The valid format of that XML data is described in a *schema* by using the Schema description Language described therein.

The IP-XACT specification can be viewed as a mechanism to express and exchange information about design IP and its required configuration. For the IP provider, the IP configuration or generator script provider, the point-tool provider, or the SoC design-tool provider to claim IP-XACT compliance, they shall adhere to the completeness and IP-XACT semantic consistency rules (SCRs) as outlined in 1.4 and Annex B.

The use of The SPIRIT Consortium specified formats and interfaces are shown in Figure 1 and described in the following subsections. The IP-XACT specifications relate directly to those aspects of the DE indicated in **bold**.

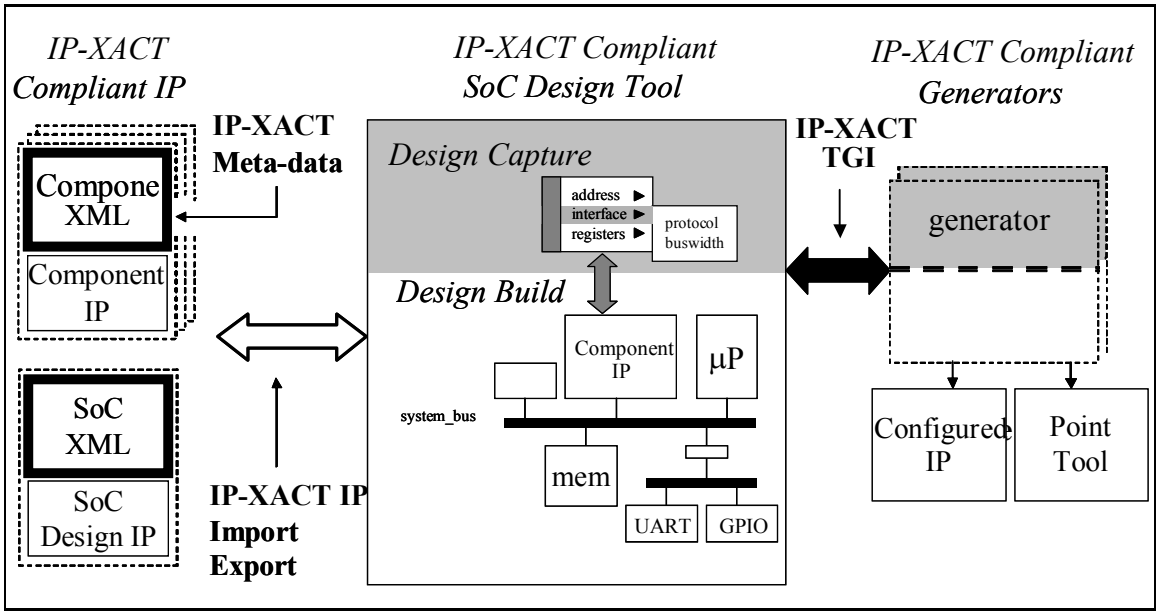


Figure 1—IP-XACT design environment

1.3.1 System design tool

System design tools enable the designer to work with IP-XACT design IP through a coordinated front-end and IP design database. These tools create and manage the top-level meta-description of system design and provide two basic types of services: *design capture*, which is the expression of design configuration by the IP provider and design intent by the IP user; and *design build*, which is the creation of a design (or design model) to those intentions.

As part of design capture, a system design tool must recognize the structure and configuration options of imported IP. In the case of *structure*, this implies both the structure of the design (e.g., how specific pin-outs refer to lines in the HDL code) as well as the structure of the IP package (e.g., where design files and related generators are provided in the packaged IP data-structure). In the case of *configuration*, this is the set of options for handling the imported IP (e.g., setting the base address and offset, bus-width, etc.) that may be expressed as configurable parameters in the IP-XACT meta-data.

As part of design build, generators are provided internally using a system design tool to achieve the required IP integration or configuration, or provided externally (e.g., by an IP provider) and launched by the system design-tool as appropriate.

The *system design tool set* defines a DE where the support for conceptual context and management of IP-XACT meta-data resides. However, the IP-XACT specifications make no requirements upon system design

tool-architecture or a tool's internal data structures. To be considered IP-XACT v1.4 enabled, a system design-tool must support the import/export of IP expressed with valid IP-XACT v1.4 meta-data for both component IP and systems, and it must support the Tight Generator Interface (TGI) for interfacing with external generators (to the DE).

1.3.2 Design intellectual property

IP-XACT is structured around the concept of IP re-use. IP may be considered from the perspective of the object itself, its supporting views, and meta-data description. In IP-XACT v1.4, the specifications need to be comprehensive for all design objects required to support ESL and RTL design and integration. These include the following:

a) Design objects
   1) TLM descriptions: SystemC & SystemVerilog
   2) Fixed HDL descriptions: Verilog, VHDL
   3) Configurable HDL descriptions (e.g., bus-fabric generators)
   4) Design models for RT and transactional simulation (e.g., compiled core models)
   5) HDL-specified verification IP (e.g., basic stimulus generators and checkers)

b) IP views—This is a list of different views (levels of description and/or languages) to describe the IP object. In IP-XACT v1.4, these views include:
   1) Design view: RTL Verilog or VHDL, flat or hierarchical components
   2) Simulation view: model views, targets, simulation directives, etc.
   3) Documentation view: specification, User Guide, etc.

1.3.3 Generators

Generators are executable objects (e.g., scripts) that may be integrated within a SoC design tool (referred to as 'internal'), or provided separately as an executable that can be launched (referred to as 'external'). Generators may be provided as part of an IP package (e.g., for configurable IP, such as a bus-matrix generator) or as a way of wrapping point tools for interaction with a SoC design tool (e.g., an external design netlister, external design checker, etc.). In IP-XACT v1.4, external generators may only use the Tight Generator Interface (TGI) (see 1.3.4). IP-XACT is neutral regarding the underlying language of a generator (e.g., Tcl/Tk, Perl, Java, C, etc.).

Generators operate upon an IP or the system design based upon a configuration request. They are launched during the build phase of a design environment, i.e., generators create the design to the specification provided in the design capture phase. Generators may perform multiple tasks, such as IP creation, configuration, post-generation checking, simulation set-up, etc. They may also be part of a configurable IP package or a specific design-automation feature, such as an architecture-specific design-rule checker. Some generation services are provided internally to SoC design tools and some specialized generation services may need to be provided externally. For IP-XACT v1.4, external generators can only operate upon IP-XACT compliant meta-data through the TGI.

Not all generators require the ability to modify the internal meta-data representation of the SoC, e.g., a generator checking build correctness may just return a pass/fail result. However, many generators do need to modify the meta-data description, even if only minor modifications occur, e.g., an IP generator will need to communicate with the SoC design tool where the generated RTL is placed.

Finally, generators can be associated with phases in the design process that enable sequencing of chains of generator chains. This is critical for providing script-based support of SoC creation and simulation.

1.3.4 IP-XACT interfaces

There are two obvious interfaces expressed in Figure 1: from the SoC Design Tool to the external IP libraries and from the SoC design Tool to the generators. In the former case, the IP-XACT specifications are neutral regarding the use of design-tool interfaces to IP repositories. While being able to read and write IP with IP-XACT meta-data is a requirement of the specification, the formal interaction between an external IP repository and a SoC design-tool is not specified.

IP-XACT v1.4 supports a TGI that is Simple Object Access Protocol (SOAP) based and Web Services Description Language (WSDL) specified. The TGI provides an efficient interface for external generators, which is required due to the generally rapid nature of generator execution. Using the language-independent, SOAP-based message passing interface, generators that are IP-XACT compliant are DE independent, i.e., a generator running on a design shall produce the same results independent of the DE in which it is run.

1.4 IP-XACT enabled implementations

Complying with the rules outlined in this section allows the provider of tools, IP, or generators to class their products as *IP-XACT Enabled*. Conversely, any violation of these rules removes that naming right. This section first introduces the set of metrics for measuring the valid use of the specifications. It then specifies when those validity checks are performed by the various classes of products and providers: DEs, point tools, IPs, and generators.

a) Parse validity
   1) Parsing correctness: Ability to read all IP-XACT files.
   2) Parsing completeness: Cannot require information which could be expressed in an IP-XACT format to be specified in a non- IP-XACT format. Processing of all information present in an IPXACT document is not required.

b) Description validity
   1) Schema correctness: IP is described using XML files that conform to the IP-XACT schema.
   2) Usage completeness: Extensions to the IP-XACT schema shall only be used to express information that cannot otherwise be described in IP-XACT.

c) Semantic validity
   1) Semantic correctness: Adheres to the semantic interpretations of IP-XACT data described in this standard.
   2) Semantic completeness: Obeys all the semantic consistency rules described in Annex B.

These validity rules can be combined with the product class specific rules to cover the full IP-XACT enabled space. The following subsections describe the rules a provider has to check to claim a product is IP-XACT Enabled.

1.4.1 Design environments

An IP-XACT Enabled design environment:

a) Shall follow the Parse Validity Requirements shown in 1.4.
b) Shall only create IP which is IP-XACT Enabled.
c) When modifying any existing IP-XACT files, shall do so without losing any pre-existing information. In particular, it shall preserve any vendor extension data included in the existing IP-XACT file.
d) Shall support the IP-XACT generator interfaces fully for interaction with underlying database.
e) Shall be able to invoke all IP-XACT Enabled generators.

1.4.2 Point tools

An IP-XACT Enabled point tool:

a) Shall follow the Parse Validity Requirements shown in 1.4.

b) Shall only create IP which is IP-XACT Enabled.

c) When modifying any existing IP-XACT files, shall do so without losing any pre-existing information. In particular, it shall preserve any vendor extension data included in the existing IP-XACT file.

1.4.3 IPs

An IP-XACT Enabled IP shall have an IP-XACT description that follows the Description and Semantic validity requirements. In addition, any generators associated with this IP shall be IP-XACT Enabled generators.

1.4.4 Generators

An IP-XACT Enabled generator:

a) Shall only create IP which is IP-XACT Enabled.

b) When modifying any existing IP-XACT files, shall do so without losing any pre-existing information. In particular, it shall preserve any vendor extension data included in the existing IP-XACT file.

c) Shall be callable though the IP-XACT generator interface.

d) Shall only communicate with the DE that invoked it through the IP-XACT generator interface.

1.5 Conventions used

Each clause which details any IP-XACT usage defines it own conventions and meta-syntax as needed. The conventions used throughout the document are included here.

1.5.1 Visual cues (meta-syntax)

Bold: shows required keywords and/or special characters, e.g., **addressSpace**. For the initial use (per element), keywords are shown in **boldface-red text**, e.g, **bitsInLau** (see also: 1.6).

Bold italics: shows group names, e.g., ***nameGroup***. **Need to do so consistently**

`Courier`: shows examples, external command names, directories and files, etc.,
e.g., address `0x0` is on `D[31:0]`.

***Add any other document conventions here*. See IEEE Std 1800-2005 for an example.**

1.5.2 Notational Conventions

The keywords "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in the IETF Best Practices Document 14, RFC-2119.

1.5.3 Syntax examples

Any syntax examples shown in this Standard are for information only and are only intended to illustrate the use of such syntax.

1.5.4 Graphics used to document the Schema

http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/ specifies the XML schema language used to define the IP-XACT XML schemas. Normative details for compliance to the IP-XACT standard is contained in the schema files. Within this document, pictorial representations of the information in the schema files *illustrate* the structure of the schema and *define* any constraints of the standard. With the exception of scope and visibility issues, the information in the figures and schema files is intended to be identical. Where the figures and schema are in conflict, the XML schema file shall take precedence.[2]

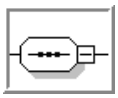
In the schema diagram figures, the various parts of the schema structure are represented graphically; the elements used to make up these figures contain much of the information contained in the schema specification. The graphics are organized into two broad categories: compositors (see 1.5.4.1) and elements (see 1.5.4.2).

1.5.4.1 Compositors in the graphic representations

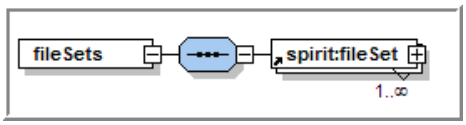
Compositors define the order in which child elements occur. There are two compositors: `sequence` and `choice`.

1.5.4.1.1 Sequence collections

A `sequence` is represented in the schema diagrams using this graphic:



An example of using `sequence`:

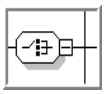


and its accompanying xml file fragment:

```
<xs:element name="fileSets">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="spirit:fileSet"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

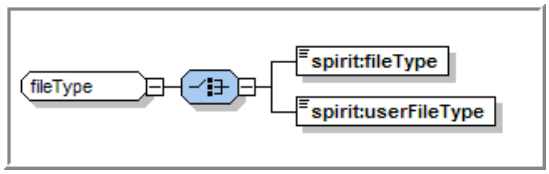
1.5.4.1.2 Choice collections

A `choice` is represented in the schema diagrams using this graphic:



[2]The graphics for this document have been generated by taking "screen-shots" of the various files as they are displayed in Altova's XML environment XMLSpy™. The use of these illustrations is not an endorsement of this tool.

An example of using `choice`:



and its accompanying xml file fragment:

```
<xs:group name="fileType">
<xs:choice>
<xs:element name="fileType">
          <xs:simpleType>
                …
          </xs:simpleType>
       </xs:element>
       <xs:element name="userFileType" type="xs:string">
</xs:choice>
</xs:group>
```

1.5.4.2 Elements of the graphic representations

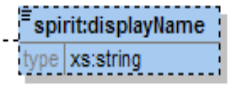
The subsequent *elements* are composed in diagrams using the constructors in 1.5.4.1. The graphical representation provides detailed information about the component's type and structural properties.

1.5.4.2.1 Mandatory single elements

The rectangle indicates an element and the solid border indicates the element is required. The absence of a number range indicates a single element (i.e., `minOcc=1` and `maxOcc=1`). **Define `minOcc`, `maxOcc`, etc.**

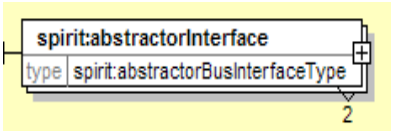
1.5.4.2.2 Single optional elements

The rectangle indicates an element and the dashed border means the element is optional. The absence of a number range indicates a single element (i.e., `minOcc=0` and `maxOcc=1`).



1.5.4.2.3 Mandatory multiple elements

The rectangle indicates an element and the solid border indicates the element is required. The number `2` means `minOcc=2` and `maxOcc=2`.

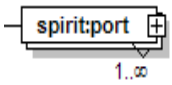


1.5.4.2.4 Optional multiple elements

The rectangle indicates an element and the dashed border means the element is optional. The number range `0..infinity` means `minOcc=1` and `maxOcc`=unbounded.

1.5.4.2.5 Mandatory multiple element containing child elements

The rectangle indicates an element and the solid border indicates the element is required. The number range `1..infinity` means `minOcc=1` and `maxOcc`=unbounded. The plus sign (+) indicates the element has complex content (i.e., at least one element or attribute child).

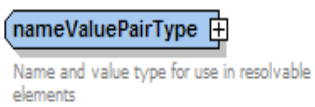


1.5.4.2.6 Elements that reference a global element

The arrow in the bottom-left means the element references a global element. The rectangle indicates an element and the dashed border indicates the element is optional. The plus sign (+) indicates the element has complex content (i.e., at least one element or attribute child). The element name in this example is `spirit:vendorExtensions`.

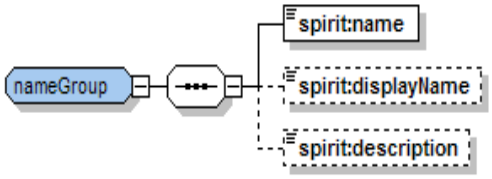
1.5.4.2.7 Complex types

The irregular hexagon with a plus sign (+) indicates an element of a complex type.



1.5.4.2.8 Macro groups

The irregular octagon with a plus sign (+) indicates a macro group. A *macro group* defines a reusable set of element declarations which are then included in schema locations with the identical effect as including the individual elements. While the diagram includes the element `nameGroup`, the actual XL documents do not include a `nameGroup` element; they can include `spirit:name` and, optionally, `spirit:displayName` and/or `spirit:description`.

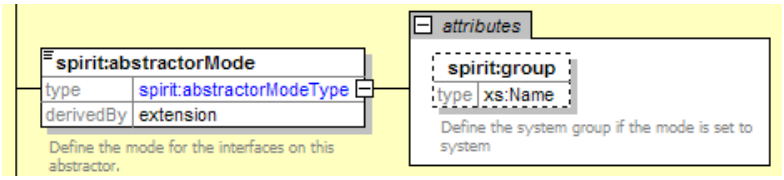


1.5.4.2.9 Wildcards

The irregular octagon with `any` at the left indicates a wildcard. Wildcards are used as placeholders to allow elements not specified in the schema or from other namespaces. `##any` elements can belong to any namespace; `##other` elements can belong to any namespace other than the ones declared in the document.

1.5.4.2.10 Attributes of an element

A rectangle with the term *attributes* (in italics) in it indicates attributes are defined for this element. Each attribute is shown in a rectangle with a dashed border.



1.6 Use of color in this standard

This standard uses a minimal amount of color to enhance readability. The coloring is not essential and does not affect the accuracy of this standard when viewed in pure black and white. The places where color is used are the following:

— Cross references that are hyperlinked to other portions of this standard are shown in underlined-blue text (hyperlinking works when this standard is viewed interactively as a PDF file).

— Syntactic keywords and tokens in the formal language definitions are shown in **boldface-red text**.

1.7 Contents of this standard

The organization of the remainder of this standard is as follows:

- Clause 2 provides references to other applicable standards that are assumed or required for this standard.
- Clause 3 defines terms and acronyms used throughout the different specifications contained in this standard.
- Clause 4 defines the interoperability use model.
- Clause 5 previews the schema and their object definitions.
- Clause 6 defines the buses and interconnect models.
- Clause 7 defines the component models.
- Clause 8 defines the designs and their connections.
- Clause 9 defines the adapters between abstraction definitions.
- Clause 10 describes generators and their use in IP-XACT.
- Clause 11 defines the design models and their configuration.
- Clause 12 previews addressing and addressing formulas.
- Annexes. Following Clause 12 are a series of annexes.

2. Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEC/IEEE 61691-1-1, Behavioral languages—Part 1: VHDL language reference manual.[1, 2]

IEEE Std 1364™, IEEE Standard for Verilog Hardware Description Language.[3]

ISO/IEC 8859-1, Information technology—8-bit single-byte coded graphic character sets—Part 1: Latin Alphabet No. 1.[4]

ISO/IEC 8879, SGML **Get exact title and call numbers.**

The IP-XACT Schema v1.4 is available from the SPIRIT Consortium web site at:
http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4/index.xsd

The IP-XACT TGI API v1.4 format is available from the SPIRIT Consortium web site:
http://www.spiritconsortium.org/releases/tgi/TGI.wsdl

SOAP (Simple Object Access Protocol) Version 1.2 is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. “Part 1: Messaging Framework” defines, using XML technologies, an extensible messaging framework containing a message construct that can be exchanged over a variety of underlying protocols:
http://www.w3.org/TR/2007/REC-soap12-part1-20070427/

Web Services Description Language (WSDL) 1.1 is used to describe the Tight Generator interface. The specification can be found at: http://www.w3.org/TR/wsdl

The XML version 1.0 is available from the W3C web site:
http://www.w3.org/TR/2000/REC-xml-20001006.

The XML Schema specification is available from the W3C web site:
http://www.w3.org/TR/2004/REC-xmlschema-0-20041028;
http://www.w3.org/TR/2004/REC-xmlschema-1-20041028;
http://www.w3.org/TR/2004/PER-xmlschema-2-20040318.

The XPath specification, version 1.0, is available from the W3C web site:
http://www.w3.org/TR/1999/REC-xpath-19991116.

The XPath version 2.0 is available from the W3C web site:
http://www.w3.org/TR/2005/CR-xpath20-20051103.

[1]IEC publications are available from the Sales Department of the International Electrotechnical Commission, Case Postale 131, 3, rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iec.ch/). IEC publications are also available in the United States from the Sales Department, American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

[2]IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).

[3]The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[4]ISO/IEC publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembé, CH-1211, Genève 20, Switzerland/Suisse (http://www.iso.ch/). ISO/IEC publications are also available in the United States from Global Engineering Documents, 15 Inverness Way East, Englewood, CO 80112, USA (http://global.ihs.com/). Electronic copies are available in the United States from the American National Standards Institute, 25 West 43rd Street, 4th Floor, New York, NY 10036, USA (http://www.ansi.org/).

The XSLT version 1.0 is available from the W3C web site:
http://www.w3.org/TR/1999/REC-xslt-19991116.

**Move this into DE compliance?? XSLT version 1.0 support is required for DE compliance, XSLT, version 2.0 is optional. For maximum portability, IP and generators should make use of version 1.0.

3. Definitions, acronyms, and abbreviations

For the purposes of this document, the following terms and definitions apply. *The Authoritative Dictionary of IEEE Standards Terms* [B2][5] should be referenced for terms not defined in this clause.

3.1 Definitions

3.1.1 abstraction definition: An object that describes a type of **bus interface**, including details of the **port**s this type of **bus interface** may have and the **constraint**s that apply to these **port**s.

3.1.2 ad-hoc connection: Directly connects two **port**s without the use of **bus interface**s or **interconnection**s. Wire ad-hoc connections have a wire protocol and cable ad-hoc connections have a cable connection.

3.1.3 abstractor: A top level IP-XACT element used to convert between two **bus interfaces** having different abstraction types and sharing the same bus type.

3.1.4 active interface: An **interface** that participates in the transactions.

3.1.5 AMBA: An open specification on-chip backbone for interconnecting **intellectual property** (IP) blocks.

AMBA is a Registered Trade Mark owned by ARM, and needs to be acknowledged as such somewhere in this document. I am not sure if the specific AMBA bus types (AHB, APB, AXI etc.) are also registered trade marks.

3.1.6 application programmers interface (API): A method for accessing design and **meta-data** in a procedural way.

3.1.7 architectural rules: Generic rules which define how **subsystems** relate to **platforms** that relate to **components** of system design.

3.1.8 behavioral properties of a memory location: The behavioral properties of a bit in memory are defined as

a) Its access rights.
b) Its volatility.
c) Whether it has a defined reset value and what this value is.
d) The width of the memory area containing it:
    1) For bits within parallel banks, this is the width of the top-level parallel bank containing it.
    2) For all other bits, this is the width of the containing address block.
e) The effective least addressable unit (i.e., the value of **bitsInLau**) of its containing memory map. Bridges between the memory location and the bus interface from which it is observed may modify a location's effective least addressable unit from is the one defined in the memory map.

[5]The number in brackets correspond to those of the bibliography in Annex A.

f) The endianness of its containing address block.

g) The usage of its containing address block

h) Its dependencies:

Two bits have the same dependencies if they depend on the same values of the same bits at the same address. Since different memory maps may vary in how they name registers and fields (and even in how they split the address spaces into registers and fields), it is possible for two dependencies to match even if they use different register and field names.

This should move to another chapter

3.1.9 bridge: A mechanism to model the internal relationship between **master interfaces** and **slave interfaces** inside a **component**. Bridges explicitly describe the internal point-to-point connections between the component interfaces. A bridge can have multiple address spaces, can be hierarchical, supports memory mapping and re-mapping, and can only have direct interfaces. *Syn*: **bus bridge**.

3.1.10 bus: A collection of **ports** used to connect blocks connected to it involving both hardware and software protocols. Within IP-XACT, buses are **components**.

3.1.11 bus definition: An **object** that describes the high-level properties for a **bus**, such as the maximum masters allowed or if one bus expands upon the definition of another.

3.1.12 bus interface: The **interface** of an **IP** to a **bus**. **Components** are connected together by linking the bus interfaces together. There are three different classes of bus interfaces: master, slave, and system with two flavors: direct and mirrored.

3.1.13 channel: A special **object** that can be used to describe multi-point connections between regular components, which may require some interface adaptation.A channel connects **component master**, **slave**, and **system interfaces** on the same **bus**. A channel can also represent a simple wiring interconnection or a more complex structure, such as a bus. A channel can only have one address space. Channel interfaces are always mirrored interfaces. A channel supports memory mapping and re-mapping.

3.1.14 component: The central place holder for object **meta-data** and its bus and generator **interfaces**. Components are used to describe cores, peripherals, and buses. Components may reference designs to create hierarchy. *Syn*: **component description**.

3.1.15 configurable component: A **component** which has some **parameter**s the DE can configure; these parameters are also configurable in the RTL or TLM model.

3.1.16 configurable IP: **IP** which has parameters and is customized by setting/configuring the parameters. There may also be IP-specific generators capable of creating new components from the configured component and updating the design with the new version of the **component**.

3.1.17 configuration manager: An object which creates and manages top-level meta-description of **system on a chip** (SoC) design. It can annotate SoC schema with details of a specific SoC design including: IP versions, IP views, IP configuration, IP connectivity, and IP constraints. It manages the launching of **IP generators** and **tool plug-ins**, and any meta-data updates occurring as a consequence of a launch. It also handles the updating and retrieval of relevant **IP meta-data** from the IP repository.

3.1.18 connection: Generally describes a communication mechanism between one or more components.

3.1.19 constraint: A constraint defines a limitation on a part of the system that needs to be satisfied for the system to be correct. Timing constraints are often specified on ports, requiring that during a given clock cycle the value of the signal become stable in a certain time period and remain stable for a certain time period relative to a particular clock edge.

3.1.20 constraint set: **Constraints** defined in groups to associate different constraints with different views of the component.

3.1.21 design: An IP-XACT description of a **system** or **subsystem** listing its **component**s, the **connection**s between these components, and the **interface**s exported by the system or subsystem.

3.1.21.1 design configuration: This file contains non-essential ancillary information for generators, the active or current view selected for instances in the design, and configurable information defined in vendor extensions. It references a design file and can specify a **view** for the **component** instances and **abstractor**s for each **interconnection**, and configure generator **chain**s. *Syn*: **configuration**.

3.1.22 design database: Working storage for both **meta-data** and **component** information that helps create and verify **systems** and **subsystems**.

3.1.23 design environment (**DE**): The coordination of a set of tools and **IP**, or expressions of that IP (e.g., models) so the system-design and implementation flows of a SoC re-use centric development flow is efficiently enabled. This is managed by creating and maintaining a meta-data description of the **SoC**.

3.1.24 endianness: **big endian** is the most significant byte at the lowest memory address and **little endian** is the least significant byte at the lowest memory address.

3.1.25 electronic system level (**ESL**) A high level of design modeling typically done with, but not limited to, SystemC or SystemVerilog design languages.

3.1.26 external components: **Components** that do not end up on the **SoC**, but are needed for total system verification.

3.1.27 fixed IP: **IP** that has no parameters which are configured by the **DE** or set by industry de-facto tools.

3.1.28 generator: Combines **component meta-data** with **architectural rules** to provide a consistent system description which uses a specified **tight generator interface** (**TGI**) to generate specific design views or **configurations** for the purposes of supporting a number of design styles. The generator may add/remove/replace components, add/remove/replace interconnections, add/remove/replace project settings, and add/remove/replace persistent data.

3.1.29 generator API: This **API** provides a common interface for algorithmic code in a **generator** or **tool plug-in** to the SOAP interface of the **TGI**.

3.1.30 generator TGI: This SOAP messaging interface connects the **generators** and **tool plug-ins** to the **design environment** (**DE**), allowing the execution of these scripts and code-elements against the SoC meta-description. The **DE** enables the registration of new generators or plug-ins, exporting **SoC meta-data** and updating that data following generator or plug-in execution, and handling generator or plug-in error conditions which relate to the meta-data description.

3.1.31 generator chain: A collection of hierarchical generators to be executed in a sequence containing generators that call other generators. A design flow can be represented by a generator chain.

3.1.32 generator group: A named **generator** that contains a sequential list of **generator invocations**.

3.1.33 generator invocation: A method of running an application at a defined phase in the **generator group** with a given number of **parameters**.

3.1.34 hierarchical child bus interface: A bus interface `IC` of component `CC` is a hierarchical child of bus interface `IP` of component `CP` if and only if `CP` contains a hierarchical view, the design file of which con-

tains a hierarchical connection with interface name `IP`, component ref `CC`, and interface ref `IC`. A hierarchical child bus interface may be a hierarchical bus interface itself.

3.1.35 hierarchical component: A **component** that has one or more **view**s which reference IP-XACT design files.

3.1.36 hierarchical descendant bus interface: A bus interface `DC` is a hierarchical descendant of bus interface `AC` if and only if `DC` is a hierarchical child of `AC` or a hierarchical child of a hierarchical descendant of `AC`.

3.1.37 hierarchical family of bus interfaces: A hierarchical family of bus interfaces is a set of bus interfaces composed of a hierarchical bus interface and all its hierarchical descendants.

3.1.38 hierarchical child component: A hierarchical child of a component `C` is any component referenced in a design of `C`.

3.1.39 hierarchical descendent component: A hierarchical descendent of a component is any hierarchical child of that component or any hierarchical child of any hierarchical descendent of the component.

3.1.40 hierarchical family of components: A hierarchical family of components is a component and all its hierarchical descendents.

> **I'm not sure about having all these 'hierarchical' definitions here, especially the 'bus interface' ones; consider moving them into Chap 5 re: hierarchy and hierarchy connections**

3.1.41 initiative: An abstract description of port modes: requires, provides, or both. Used for transactional level modeling.

3.1.42 interconnection: Defines the point-to-point **connection** between two **bus interfaces**.

3.1.43 interface: A way to connect a **component** to the outside world—either **bus interfaces** or **ports**.

3.1.44 interface connection: Component interfaces with **bus definition**s and **abstraction definition**s can be listed in the design as connected to another compatible interface on another component. The listing of the **interconnection** creates a **connection** to that **interface**.

3.1.45 intellectual property (**IP**): Property utilized in the context of a SoC design or design flow, including specifications; design models; design implementation description; verification coordinators, stimulus generators, checkers and assertion / constraint descriptions; soft design objects (such as embedded software and real-time operating systems); design and verification flow information and scripts. IP-XACT distinguishes between fixed IP, parameterized IP, and configurable IP.

3.1.46 IP generators: Tools which create specific **IP** based upon **SoC meta-data** details entered into the **configuration manager**. IP generators serve as **interfaces** to IP repository for placing and retrieval of IP. and can annotate completion details (e.g., generated IP or failure of generation of IP) back into the configuration manager.

3.1.47 IP integrator: A party in the design process who receives configured IP and subsystems and combines them into a larger system.

3.1.48 IP platform architect: Creator of platform-based architectures.

3.1.49 IP provider: Creator and supplier of **IP**.

3.1.50 IP repository: Database of IP.

3.1.51 leaf component: **Components** that do not contain other IP-XACT **IP**.

3.1.52 legacy IP: **IP** that has no specific IP-XACT **meta-data view**.

3.1.53 master interface: The **bus interface** that initiates a transaction (like a read or write) on a **bus**.

3.1.54 memory map: Organization of memory elements as seen from a master interface when no memory range transformations are made, e.g., in bus bridges. Within IP-XACT, three different methods are used: a memory map at **channels**, at **transparent bridges**, or at **opaque bridges**.

3.1.55 meta-data: A tool-interpretable way of describing the design-history, locality, object association, configuration options, constraints against, and integration requirements of an **object**.

3.1.56 meta IP: **Meta-data** description of an **object**.

3.1.57 mirror interface: Has the same (or similar) **ports** to its related direct **bus interface**, but the port directions are reversed. So, a port that is an input on a direct bus interface would be an output in the matching mirror interface.

3.1.58 monitor interface: An **interface** used in **verification** that is neither a **master**, **slave**, nor **system interface**.

3.1.59 multi-layer buses: **Bus**es that have to be modeled as **component bridge**s with direct interfaces or as a hierarchical component.

3.1.60 objects: Those XML document types listed in the schema `index.xsd`: **components**, **designs**, **bus definitions**, **abstraction definitions**, **abstractors**, and **generators**. To be able to be uniquely referenced, each object has an unique identifier called its **Vendor Library Name Version** (**VLNV**).

3.1.61 opaque bridge: A bus interconnect that may modify the address.

3.1.62 Open SystemC Initiative (**OSCI**): An independent non-profit organization composed of a broad range of companies, universities and individuals dedicated to supporting and advancing SystemC as an open source standard for system-level design (see [B7])

3.1.63 parameter: Used to statically characterize (or configure) the **IP**. Parameters can be configured by the **DE** and are also configurable in the models.

3.1.64 parameterized IP: **IP** with **parameters** that can be handled by industry de-facto tools.

3.1.65 phantom port: A direction or initiative of a port which indicates this port does not have a true connection to the implementation, e.g., the port does not appear on the VHDL entity.

3.1.66 phase number: Define the sequence in which **generators** should be fired.

3.1.67 platform: Architectural (sub)system framework.

3.1.68 platform consumer: User/group who builds a **SoC** based on a particular **platform**.

3.1.69 platform provider: User/group that develops and delivers **platforms** to **platform consumers**.

3.1.70 platform rules: Rules that define how **components** interface to a specific **platform**.

3.1.71 port: Specifies interface items of a component. These interface items allow dynamic exchange of information. Connections between ports may be specified by using **ad-hoc connection**s or by including them in **bus interface**s connected together by **interconnection**s.

3.1.72 programmers view (**PV**): A level of ESL design.

3.1.73 programmers view with timing (**PVT**): A level of ESL design.

3.1.74 schema: A means for defining the structure, content, and semantics of **Extensible Markup Language** (**XML**) documents.

3.1.75 schema API: This **API** allows the **configuration manager** to query the **XML IP meta-data**. Queries may be for the existence of IP, the structure of IP, or features offered by that IP, such as configurability and interface protocol support. This **API** is also used for the import and export of meta-data when an IP block is extracted from, or imported back into, the IP management system

3.1.76 semantic rules: Additional rules applied to an XML description that cannot be expressed in the **schema**. Typically, these are rules between elements in one of multiple XML files.

3.1.77 slave interface: The **bus interface** that terminates or consumes a transaction initiated by a **master interface**. Slave interfaces often contain information about the registers accessible through the slave interface.

3.1.78 system on chip (**SoC**): Also refers to a general system which may not be implemented on a chip, such as a **transaction-level modeling** (**TLM**) design.

3.1.79 SoC platform: The top netlist containing all the instances and **connections** of the **design**.

3.1.80 style sheets: How documents are presented on screens and in print.

3.1.81 subsystem: A set of connected **components** that have dependencies on other **IP**.

3.1.82 system: A configured set of connected **components**.

3.1.83 system interface: An **interface** that is neither a **master** nor **slave interface**, and allows specialized (or non-standard) connections to a bus (e.g. clock).

3.1.84 task-level interface (**TLI**): Used for streaming interfaces between software and hardware.

3.1.85 tight generator interface (**TGI**): Used to manipulate values of elements, attributes, and parameters for IP-XACT compliant XML.

3.1.86 transaction-level modeling (**TLM**): An **abstraction level** higher than register transfer level (RTL), used for specifying, simulating, verifying, implementing, and evaluating **SoC designs**.

3.1.87 tool plug-ins: Tools which integrate **IP**, based upon **SoC meta-data** details, and prep **IP** for animation (e.g., simulation or emulation), optimization (e.g., synthesis) and **verification** (e.g., regression-suite generation). They can also annotate completion details (e.g., integrated SoC IP or failure of integration) back into the **configuration manager**.

3.1.88 transactional port: A **port** that has a service name (which can specify the data type of the port) and a port initiative. Used for high-level modeling.

3.1.89 transparent bridge: A bus interconnect that does not modify the address; it just decodes the address (by default).

3.1.90 use model: A process method of working with a tool.

3.1.91 user interface: Methods of interacting between a tool and its user.

3.1.92 validation: Proving the correctness of construction of a set of **components**.

3.1.93 verification: Proving the behavior of a set of connected **components**.

3.1.94 view: An implementation of a component. A **component** may have multiple views, each with it's own function in the design flow.

3.1.95 verification IP (**VIP**): **Components** included in a **design** for **verification** purposes.

3.1.96 Vendor Library Name Version (**VLNV**): Each IP-XACT **object** is assigned a unique identifier that is defined in the header of each XML file.

3.1.97 wire port: A **port** that describes binary values or an array of binary values. Wire ports can have a direction: in, out, or inout.

3.1.98 wire connections: **Connections** that connect **wire ports**.

3.1.99 white box interface (**WBI**): Internal points in the **IP** to be probed or driven by verification tools and/or test benches.

3.1.100 Extensible Markup Language (**XML**): A simple, very flexible text format derived from SGML (ISO/IEC 8879). **Reference this in Chap 2**

3.1.101 Xpath: An expression language used by **XSLT** to access or refer to parts of an XML document.

3.1.102 XSL: A language for expressing style-sheets and transforming XML data into HTML.

3.1.103 XSLT: A language for transforming XML documents into other types of documents.

3.1.104 3 levels of meta-data (**3MD**): This phrase refers to a hierarchy of meta-data used to support platform-based SoC architectures. The lowest level defines **IP parameters** and **constraints** and is known as the IP-level. The second level is known as the platform-level; it can be used to further constrain and capture platform rules for all SoC derivatives. The third level is the chip-level, used for any system, production, and verification tests needed to be captured for re-use and reproducibility.

3.2 Acronyms and abbreviations

AHB	AMBA high speed bus
API	application programmers interface
DE	design environment
EDA	electronic design automation
ESL	electronic system level

HDL	hardware description language
IP	intellectual property
LAU	least addressable unit (of memory)
OSCI	Open SystemC Initiative
PV	programmers view
PVT	programmers view with timing
RTL	register transfer level (design)
SCR	semantic consistency rule
SoC	system on chip
TGI	tight generator interface
TLI	task level interface
TLM	transaction-level modeling
VIP	verification IP
VLNV	Vendor Library Name Version
WBI	white box interface
XML	Extensible Markup Language
3MD	3 levels of meta-data

4. Interoperability use model

To introduce the use-model for the IP-XACT specifications, it is first necessary to identify specific roles and responsibilities within the model, and then relate these to how the IP-XACT specifications impact their interactions(s). All or some of the roles can be mixed within a single organization,e.g., some EDA providers are also providing IP, a component IP provider can also be a platform provider, and an IP system design provider may also be a consumer.

4.1 Roles and responsibilities

For this User Guide, the roles and responsibilities are restricted to the scope of IP-XACT v1.4 HDL and TLM system design.

4.1.1 Component IP provider

This is a person, group or company creating IP components or subsystems for integration into a SoC design. These IPs can be hardware components (processors, memories, buses, etc.), verification components, and/or hardware-dependent software elements. They may be provided as source files or in a compiled form (i.e., simulation model). An IP is usually provided with a functional description, a timing description, some implementation or verification constraints, and some parameters to characterize (or configure) the IP. All these types of characterization data may be described as meta-data compliant with the IP-XACT Schema. Those elements not already provided in the base schema can be provided using name-space extensibility mechanisms of the specification.

The IP provider can use one or more EDA tools to create/refine/debug IP. During this process, the IP provider may export and re-import his design from one environment to another. The IP-XACT IP descriptions need to enable this exchange for component IP.

At some point, this IP can be transferred to customers, partners and external EDA tool suppliers by using IP-XACT compliant XML. IP can be characterized into different types.

— *Fixed IP* is IP that is straightforward to describe and exchange as there are no configurable parameters. No generators need to be provided. An example of a fixed-IP is an APB GPIO block with a fixed base address.

— *Parameterized IP* are those IP blocks that do not need IP specific generators, but have ‘standard’ customizations (where ‘standard’ is defined as industry de-facto tool support), i.e., no generators need be provided for SoC design tools that support these parameterizations. An example of a parameterized IP is an AHB / APB bridge with configurable bus-widths.

— *Configurable IP* is IP created or modified as a direct result of running an IP-specific generator to build the IP to the user’s specified configuration. This IP usually requires generators to be provided with it. An example of a configurable IP is an AHB bus fabric component which has selectable number of masters and slaves, and automatic generation of decode functionality.

4.1.2 SoC design IP provider

This is a person, group or company that integrates and validates IP provided by one or more IP providers to build system platforms, which are complete and validated systems or sub-systems. Like the IP provider, the platform provider can use EDA tools to create/refine/debug its platform, but at some point the IP needs to be exchanged with others (customers, partners, other EDA tools, etc.). To do so, the platform IP has to be expressed in the IP-XACT specified format as a hierarchical component.

4.1.3 SoC design IP consumer

This is a person, group or company that configures and generates system applications based on platforms supplied by SoC Design IP providers. These platforms are complete system designs or sub-systems. Like the platform provider, the platform consumer can use EDA tools to create/refine/debug its system application and/or configure the design architecture. To do so, the EDA tool needs to support any platform IP expressed in the IP-XACT specified format.

4.1.4 Design tool supplier

This is a group or company that provides tools to verify and/or implement an IP or platform IP. There are three major tools (which could be combined) provided in a system flow:

— Platform builder (or System Design Environment) tools: these help to assemble a platform with some automation (e.g., automatic generation of interconnect).

— Verification point-tools: these handle functional and timing Simulation, Verification, Analysis, Debugging, Co-simulation, Co-verification, and acceleration.

— Implementation point-tools: these handle Synthesizing, Floorplaning, Place and routing, etc.

The EDA provider needs to be able to import IP-XACT component or system IP libraries from multiple sources and export them in the same format.

Further, IP-XACT EDA tools need to recognize, associate and launch generators that may be provided by a Generator or IP provider in support of configurable IP bundles. The imported IP might need to be created and/or modified by the tool and then exported back (e.g., to be exchanged with other EDA vendor tools) to satisfy the customer design flow.

To further support any generators supplied with IP bundles, the IP-XACT DE tools need to be able to recognize and interface with generator-wrapped point-tools. These may be provided by another EDA provider or by the IP designer/consumer as part of a company's internal design and verification flow. In general, these will support specialized design-automation features, such as architectural-rule checking.

4.2 IP-XACT IP exchange flows

This section describes a typical IP exchange flow that the IP-XACT specifications technically support between the roles defined in 4.1. By way of example, the following specific exchange flow can benefit from use of the IP-XACT specification.

> The Component IP provider generates an IP-XACT XML package and sends it to a SoC design-tool (EDA tool supplier) or directly to a Platform (i.e., SoC Design IP) provider. The EDA tool supplier imports IP-XACT XML IP and generates platform IP and/or updates (configures) the IP components. The Platform provider generates a configurable platform IP and exports it in IP-XACT XML format, which the end-user imports to build system applications. The platform provider can also generate its own platform IP into IP-XACT format and send it to the EDA provider.

Although many different possible IP exchange flows exist, from the user's viewpoint, there are three main use models:

— IP (Component or SoC Design) provider use model

— Generator (IP provider and Design tool provider) use model

— SoC design-tool provider use model

4.2.1 Component or SoC design IP provider use model

The IP provider (a hardware component IP designer or platform IP architect) can use IP-XACT to package IP in a standard and reusable format. The first step consists in creating an IP-XACT XML package (XML plus any IP views) to export the IP database in a valid format. To express this IP as an IP-XACT IP, the IP provider needs to parse the entire design file tree (which is composed of files of different types: HDL source files, data sheets, interfaces, parameters, etc.) and convert it into an IP-XACT XML format. This can be a manual step (by directly editing IP-XACT compliant XML) or an automated one (using scripts to generate Schema compliant IP-XACT XML).

Once the IP has been packaged in an IP-XACT format, the IP provider can use a SoC design-tool to write/debug/simulate/implement the IP.

4.2.2 Generator provider use model

The author of a generator expects to interact with the SoC design tool through a fixed interface during well defined times in the design life-cycle: when components are instantiated or modified or when a generator chain is started.

Generators are used within the SoC design-tool to extend its capabilities: wrapping a point tool, e.g. a simulator; wiring up IP within the design; or checking the design is correct or maybe modifying the design. Many of these features may be supplied by the IP author and handled by generators embedded in the IP itself.

Consequently, there are at least two groups of generator providers: the IP vendor, who supplies generators that are written specifically to support their IP and generic generator authors who wish to extend the features available within the SoC design-tool. This latter group will be mainly SoC-design tool vendors at first, but will also come to include third-party generator vendors.

4.2.3 System design tool provider use model

This is the chunk of the use model which needs the most expansion, TBD later.

The system design-tool takes an IP-XACT component or SoC design as input, configures it, and loads it into its own database format. Then it can automate some tasks, such as creating the platform, generating the component interconnect the bus fabric, and generating or updating the IP-XACT IP as an output (by providing new or updated XML with the attached information: new source files, parameters, documentation, etc.).

Customer design flows are usually composed of a chain of different tools from the same or different EDA vendors (e.g., when an EDA provider is not providing the entire tool chain to cover all the user flow or the customer is selecting the best-in-class point tools). To address this requirement, the EDA vendor providing an IP-XACT enabled tool needs to read and produce the IP-XACT specified format, and utilize and implement the interfaces defined by The SPIRIT Consortium. In this use model, each SoC design-tool uses its own generators (utilizing the IP-XACT TGI) to build and update its internal meta-data state in an IP-XACT format. Then the IP-XACT file can be imported by another IP-XACT enabled EDA tool.

5. IP-XACT schema

In addition to the in-line documentation for the IP-XACT Schema [B5], this chapter explains how the different schema files link to each other and when to use them.

5.1 Schema overview

The IP-XACT schema is composed of a set of main files representing the top elements (the root objects defined in 5.2) and sub files included from the main files.

NOTE—All these schema files are included by reference in the top-level schema file, `index.xsd`. IP-XACT files and DEs should reference `index.xsd` as the schema, rather than referencing the individual schema files described here.

5.1.1 Design schema

This schema defines the way in which designs can be described. A design includes instances of IP components and the interconnections between these components. The IP-XACT design schema file is called `design.xsd`.

5.1.2 Design configuration schema

This schema defines the way in which a specific configuration of a design can be described. A design includes instances of IP components and the interconnections between these components. The IP-XACT design schema file is called `design.xsd`.

5.1.3 Component schema

The component schema defines the description of an IP component. Typically, an IP component defines bus interfaces, memory maps, sub-instances, configuration information, file sets, port lists, and generators. The IP-XACT component schema file is called `component.xsd`.

5.1.4 Bus definition schema

A bus definition describes those elements of a bus that are true for all levels of abstraction. This definition also serves as a point of reference for the abstraction definitions. The IP-XACT bus definition schema file is called `busDefinition.xsd`.

5.1.5 Abstraction definition schema

An abstraction definition describes the ports that make up a bus and some expected values for port widths and usage (e.g., the `ADDR` pins can be defined as carrying address information and 16 bits wide). There is also information on expected port directions when the port is on a master, slave, or system interface. The IP-XACT abstraction definition schema file is called `abstractionDefinition.xsd`.

5.1.6 Abstractor schema

This schema defines the way that an abstractor is defined. An abstractor is a meta-design element which provides for interconnection between two abstraction definitions of the same bus definition. The abstractor can be chosen by the DE if not specified, or it can be specified in the design configuration document. The IP-XACT abstractor schema file is called `abstractor.xsd`.

5.1.7 Generator schema

These schema files define how generators are described and interact with the design environment. The IP-XACT generator schema files is called `generator.xsd`.

5.2 IP-XACT objects

The IP-XACT schema is the core of the IP-XACT specification. An IP-XACT IP appears as two distinct objects: the top-design SoC object and the Component object instantiated in the top design.

--> missing abstractor and designConfig schema

5.2.1 Object interactions

The following types of objects are those listed in the schema `index.xsd` file. See also Clause 3.

- meta-data
- bus definitions
- abstraction definitions
- components
- designs
- abstractors
- generator chains
- design configurations

The links (reference calls) between these objects is illustrated in Figure 2. The arrows (A → B) illustrate a reference of object `B` from object `A`.

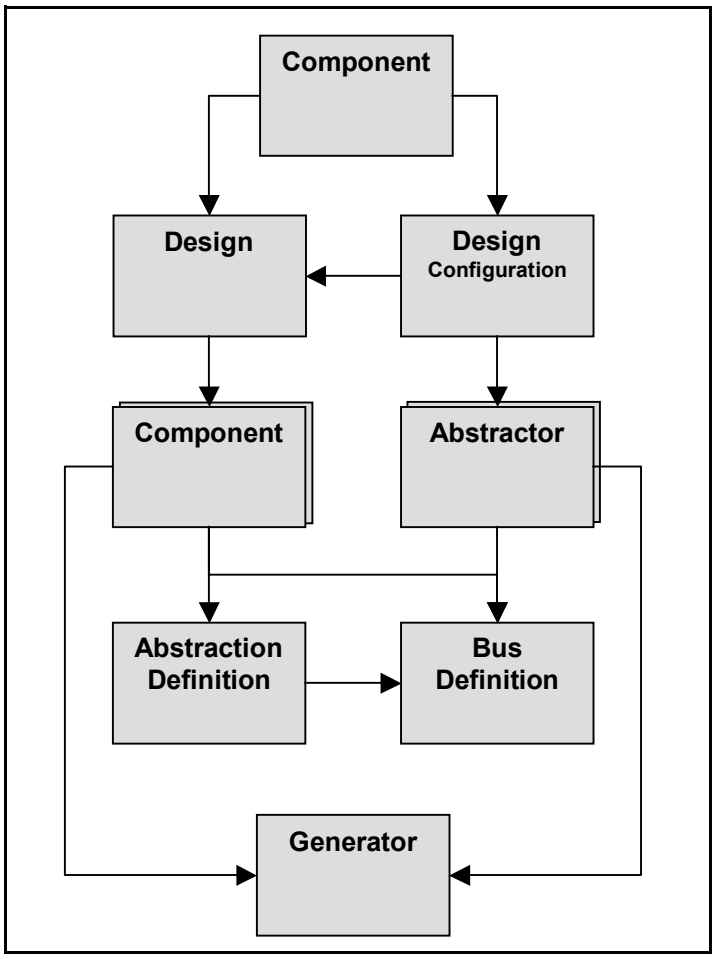


Figure 2—IP-XACT object interactions

--> Figure 2 is not correct w.r.t. its definition. If it is intended to show the VLNV relationship, the generator (**generatorChain** to be exact) should only be referenced from **designConfig**.

To be uniquely referencable, each of these objects has a unique identifier in IP-XACT terms, called a Vendor Library Name Version (*VLNV*).

5.2.2 VLNV

Each object is assigned a VLNV that is defined in the header of each XML file, e.g.,

```
<spirit:vendor>spiritconsortium.org</spirit:vendor>
<spirit:library>Leon2</spirit:library>
<spirit:name>simple_design</spirit:name>
<spirit:version>1.0</spirit:version>
```

The VLNV is used as a unique identifier in an design environment. Only one object with a given VLNV may be present in a design environment at any given time. The timing and way to change the VLNV of an object is completely up to the user or developer.

The `vendor` (first V of VLNV) element shall be the domain name of the organization responsible for the object (e.g., `spiritconsortium.org`). This need not be the owner or creator of the IP described by the object. If company `XYZ` creates a object, the vendor element shall be set to their domain name, which could be `xyz.com`.

The `version` number (last V of the VLNV) assigned to any object may be more complex than an integer number. The version number may appear as an alphanumeric string and contain a set of substrings, with non-alphanumeric delimiters in-between. Each IP supplier shall have their own cataloguing system for setting version numbers.

5.2.2.1 Sorting and comparing

Sorting and comparing a VLNV string determines whether:

— an IP is a component that has been previously imported;

— multiple versions of the same IP can exist in a design.

To sort and compare the VLNV, subdivide the version number into major fields and subfields. Major fields may be separated by a non-alphanumeric delimiter such as `/`, `.`, `-`, `_`, etc. Each major field can be compared to determine equivalence and broken down further into subfields if necessary.

5.2.2.2 Comparison rules

a) Each version number is broken into its major fields, which are separated using the appropriate delimiter (e.g., / or .)

b) Major fields are compared against each other from left-to-right.

c) Subfields, within each major field, need to be examined if the major fields are alphanumeric. Each major field shall have alphabetical and numerical subfields that are separated from right-to-left.

d) To summarize the rules for the comparison of each subfield in a major field:

   1) Numeric—compare the integer values of numeric subfields.

   2) Alphabetic:

      i) String—perform a simple string comparison.

      ii) Case—ignore alphabetic case (e.g., `a-A` are the same).

There are a few cases where the version numbers are considered as equal, but this may not be obvious to the user. For example, under these rules, `A1` and `A01` are equal, since numerical subfields are compared numerically, and `A.B` equals `A_B`, since delimiters are not compared.

5.2.2.3 Examples

The following examples illustrate the sorting and comparing of a VLNV.

Example 1

The first case uses: `205/75WR16` and `215/50HR15`.

a) Each of these version numbers break down into the following two major fields, separated by the `/` delimiter: `205 75WR16` and `215 50HR15`.

b) Major fields are compared against each other from left-to-right. In this example, the first major fields (`205` and `215`) differ between the VLNV strings and the comparison ends there. This case is also simplified since the first major field is an integer (i.e., numeric).

c) Subfields, within each major field, need to be examined if the major fields are alphanumeric. Each major field shall have alphabetical and numerical subfields that are separated from right-to-left.

Example 2

In the next example, two VLNV have the same first major field, their second major subfields need to be compared: e.g., `205/45R16` and `205/55R15`.

a) The first major field (`205`) is equal between these two VLNV so the second major field is checked. These second major fields are broken down into the following alphabetic and numeric subfields: `45 R 16` and `55 R 15`.

b) The subfields are compared from left-to-right. The first (and in this case only) comparison is `45` versus `55`, so these subfields are not equal. The major fields are not equivalent.

5.2.3 Version control

Each file conforming to the top-level schema has a set of VLNV elements which, when considered together, form a unique identifier (a *version control number*) for the information contained in that XML document. The VLNV of any IP-XACT information is not the same as the version of the file which might contain that information.

NOTE—A XML file might be revised in a way that does not materially affect the IP-XACT information content. For example, copyright notices are updated, comments are added, and environment variable names used as part of the filenames might be changed (but still point to the same files). These changes do not necessitate changing the VLNV.

Many developers of IP libraries use a version control system to track updates and changes to the various files that contribute to the overall design and IP package information. At any time, individual files may be modified and updated as development of that design or IP progresses. At appropriate junctures, releases are made, each consisting of a particular combination of files at different levels of version.

An IP-XACT file is one of the files that can be very usefully tracked in this way and updated in-line with other design modifications. There is no direct link between the version number of the file and the VLNV identifier contained in that file. In many, but by no means all cases however, the VLNV will be coordinated with the overall release package version.

5.3 Design models

An *IP-XACT design* is a description that contains all instances and connections of the design. The following sections have to be defined in the design:

— the VLNV of this IP
— the component instances (e.g., core, peripherals, and buses)
— the connections between the component instances.

Figure 3 illustrates a simple IP-XACT platform design.

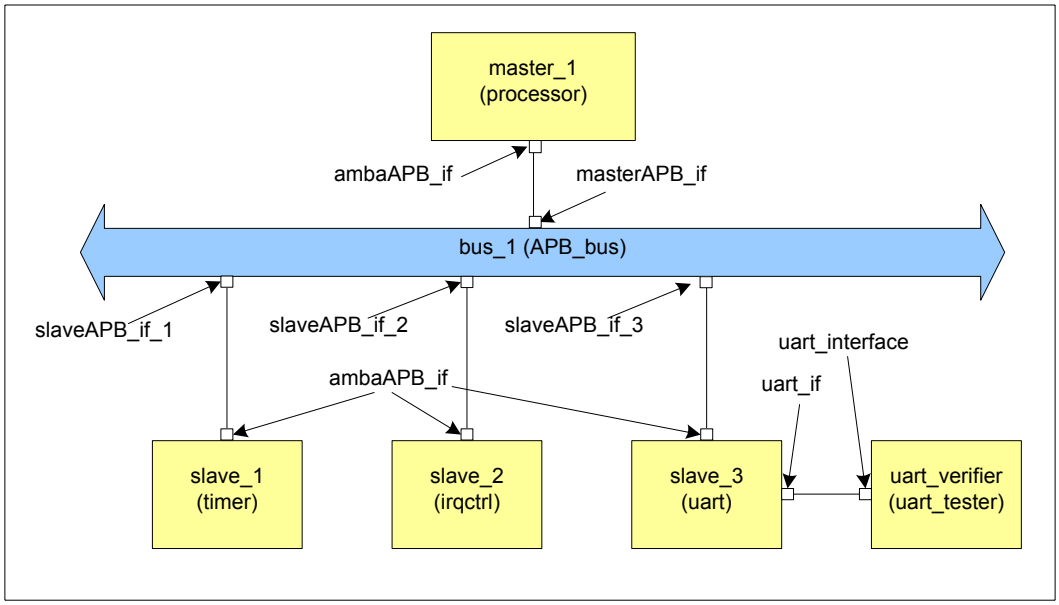


Figure 3—Simple SoC design example

The equivalent XML file for this simple design is described in the remainder of this chapter. The rest of this Standard defines the IP-XACT elements and attributes used in building this design and its sections.

5.3.1 Design

The design starts with the standard XML headers and includes the design's VLNV, there's then a list of components, followed by a list of interconnections, as shown in the following XML fragment.

```
<?xml version="1.0" encoding="UTF-8" ?>
<spirit:design
xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4
    http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4/index.xsd">
  <spirit:vendor>spiritconsortium.org</spirit:vendor>
  <spirit:library>simple_lib</spirit:library>
  <spirit:name>simple_design</spirit:name>
  <spirit:version>1.0</spirit:version>
  <spirit:componentInstances>
<spirit:interconnections>
</spirit:design>
```

5.3.2 Hierarchy represented by a design file

Hierarchical designs can be described in IP-XACT. In any IP-XACT design, the design file references components files. In a hierarchical design, some or all of these component files have views which reference further design files or design configuration files describing the design of those components, as depicted in Figure 4. This linking allows for unlimited levels of hierarchy in a design. All referencing of designs and configurations of designs and components in IP-XACT are done through the VLNV (see 5.2.2). Four elements (`vendor`, `library`, `name`, and `version`) uniquely identify a design, a configuration of a design or a component.

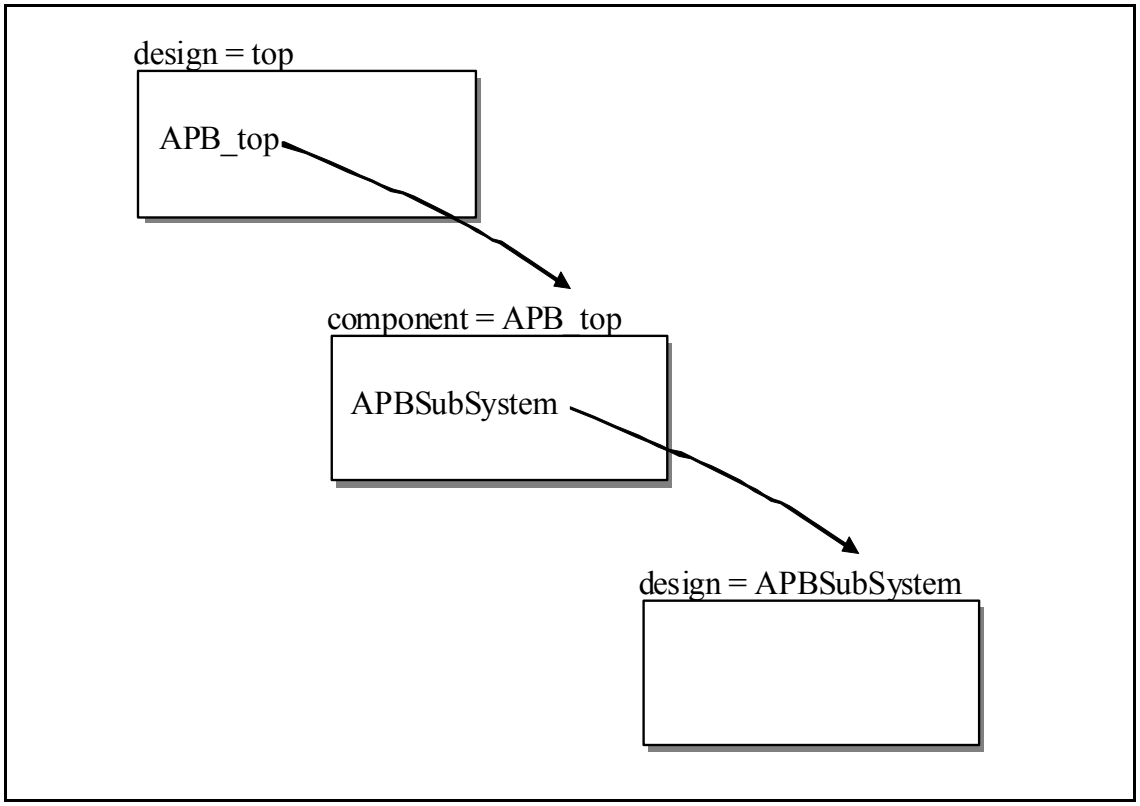


Figure 4—Hierarchy example

This is an example of the highest `Design` file in a hierarchical design.

```
<spirit:design>
  <spirit:vendor>spiritconsortium.org</spirit:vendor>
  <spirit:library>Example</spirit:library>
  <spirit:name>Top</spirit:name>
  <spirit:version>1.00</spirit:version>
…
  <spirit:componentInstance>
    <spirit:instanceName>APB</spirit:instanceName>
    <spirit:componentRef
      spirit:vendor="spiritconsortium.org"
      spirit:library="Example"
      spirit:name="APB_top"
      spirit:version="1.00" />
  </spirit:componentInstance>
```

This is an example of a `Component` file in a hierarchical design.

```
<spirit:component>
  <spirit:vendor>spiritconsortium.org</spirit:vendor>
  <spirit:library>Example</spirit:library>
  <spirit:name>APB_top</spirit:name>
  <spirit:version>1.00</spirit:version>
…
  <spirit:model>
    <spirit:views>
```

```
        <spirit:view>
          <spirit:name>Hierarchical</spirit:name>
            <spirit:envIdentifier>::</spirit:envIdentifier>
               <spirit:hierarchyRef
                 spirit:vendor="spiritconsortium.org"
                 spirit:library="Example"
                 spirit:name="APBSubSystem"
                 spirit:version="1.2"/>
        </spirit:view>
```

This is an example of the lower `Design` file in a hierarchical design, showing the hierarchical connection of a bus interface.

```
<spirit:design>
  <spirit:vendor>spiritconsortium.org</spirit:vendor>
  <spirit:library>Example</spirit:library>
  <spirit:name>APBSubSystem</spirit:name>
  <spirit:version>1.2</spirit:version>
…
    <spirit:hierConnections>
      <spirit:hierConnection spirit:interfaceRef="UartIF1">
      <spirit:activeInterface spirit:componentRef=”slave_3”
            spirit:busRef=”uart_if”></spirit:activeInterface>
        </spirit:hierConnections>
    </spirit:hierConnections>
```

5.3.3 Design interconnections

Design interconnections (**interConnection**s between active interfaces and **monitorInterconnection**s between active and monitor interfaces) can be given a name, as illustrated in Figure 5.

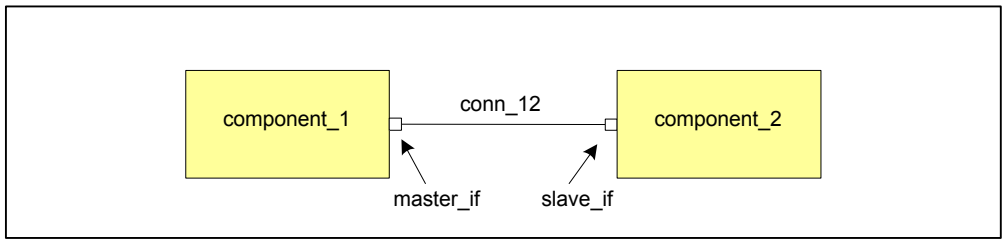


Figure 5—Connectivity name example

These interconnections could be built using the following XML fragment.

```
<spirit:interConnections>
   <spirit:interConnection>
      <spirit:name>conn_12</spirit:name>
      <spirit:activeInterface
                        spirit:componentRef="component_1"
                        spirit:busRef="master_if"/>
      <spirit:activeInterface
                        spirit:componentRef="component_2"
                        spirit:busRef="slave_if"/>
   </spirit:interConnection>
</spirit:interConnections>
```

This fragment illustrates the connectivity between the bus interface `master_if` (on `component_1`) and the bus interface `slave_if` (on `component_2`) in the design shown in Figure 5. The DE (or the user) can name this connection (e.g., `conn_12`). This name is optional, but if defined, it shall be unique for all **interConnections** elements inside the design.

This last sentence shows semantics; add a xref here or move this to **interConnections??

5.3.4 Hierarchical connectivity

In IP-XACT, hierarchical connectivity can also be expressed in the design file, as shown in Figure 6.

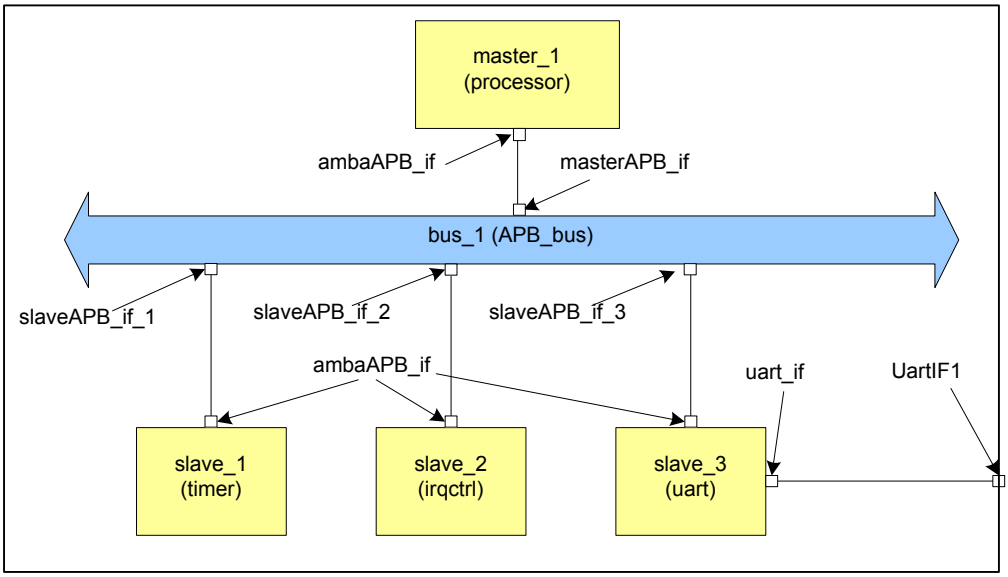


Figure 6—Hierarchical connectivity example

These hierarchical connections could be built using the following XML fragment.

```
<spirit:hierConnections>
  <spirit:hierConnection spirit:interfaceRef="UartIF1">
    <spirit:activeInterface spirit:componentRef=”slave_3”
         spirit:busRef=”uart_if”/>
     </ spirit:hierConnection>
</spirit:hierConnections>
```

This fragment illustrates the connectivity between the bus interface `UartIF1` (on the component that is being described by this design) and the bus interface `uart_if` on the UART instance `slave_3` in the design shown in Figure 6. The DE needs to ensure the interface `UartIF1` exists on the component when referencing a design file from a component.

This last sentence and the following Note show semantics; add a xref here or move these to **hierConnections??

NOTE—A bus cannot be hierarchically connected, this would require splitting the bus component. However, it is possible to connect an interface of a bus via hierarchical connection to a bus on a higher level. In most cases, this is done via an additional bus bridge.

6. Interface definition descriptions

6.1 Definition descriptions

In IP-XACT, a group of ports that together perform a function are described by a set of elements and attributes split across two definitions, an bus definition and an abstraction definition. These two descriptions are referenced by components or abstractors in their bus interfaces.

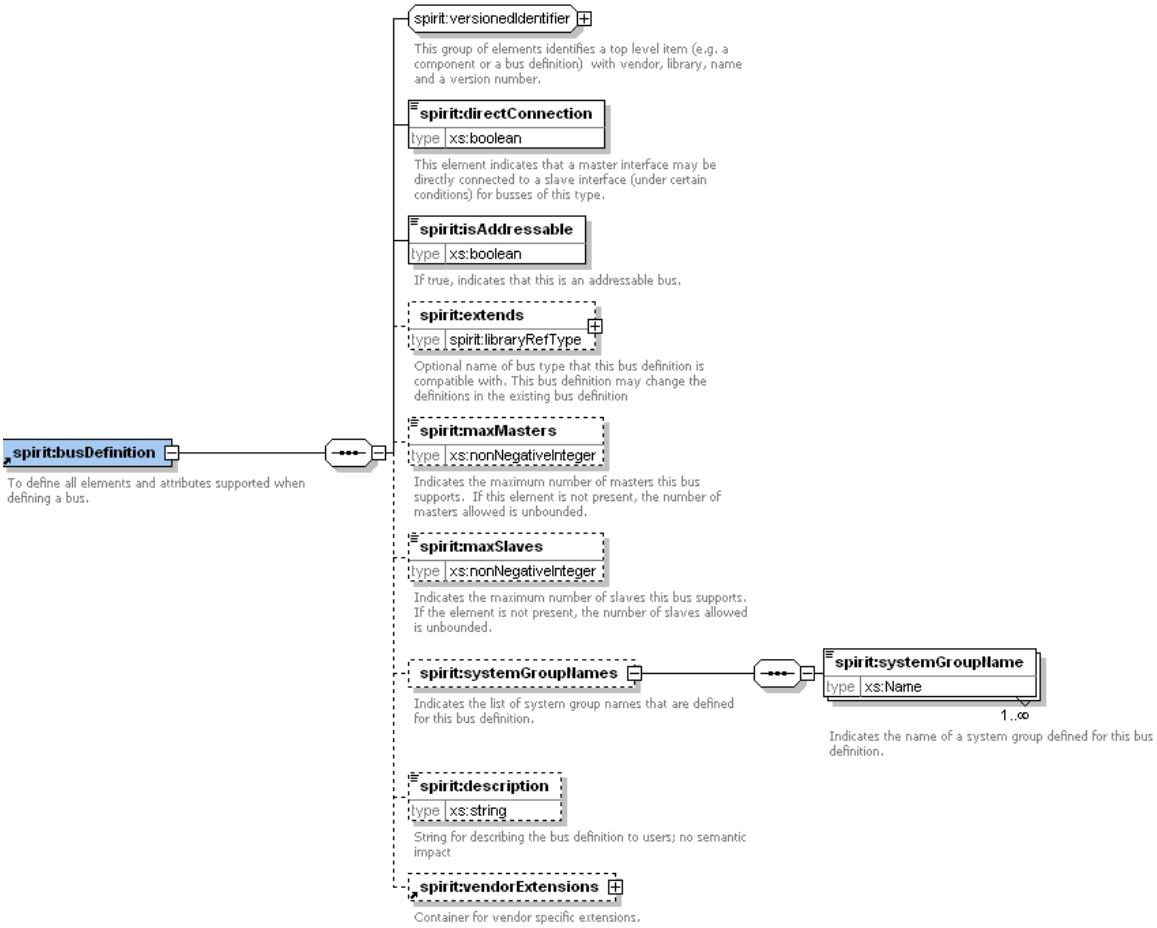
The *bus definition* description contains the high-level attributes of the interface, including items such as the connection method and indication of addressing. 7.5 describes bus interfaces.

The *abstraction definition* contains the low-level attributes of the interface, including items such as the name, direction, and width of the ports. This is a list of logical ports that may appear on a bus interface for that bus type.

6.2 Bus definition

6.2.1 Schema

The following schema details the information contained in the **busDefinition** element, which is one of the seven top-level elements in the IP-XACT specification used to describe the high-level aspects of a bus.



6.2.2 Description

The top-level **busdefinition** element describes the high-level aspects of a bus or interconnect. It contains the following elements and attributes.

a) The ***versionedIdentifier*** group provides a unique identifier; it consists of four subelements for a top-level IP-XACT element.
   1) **vendor** (mandatory) identifies the owner of this description. The recommended format of the **vendor** element is the company internet domain name.
   2) **library** (mandatory) identifies a library of this description. This allows one vendor to group descriptions.
   3) **name** (mandatory) identifies a name of this description.
   4) **version** (mandatory) identifies a version of this description. This allows one vendor to provide many descriptions which all have the same name, but are still uniquely identified.

b) **directConnection** (mandatory) specifies what connections are allowed. The **directConnection** element is of type ***Boolean***. A value of ***True*** specifies these interfaces may be connected in a direct master to slave fashion. A value of ***False*** indicates only non-mirror to mirror type connections are allowed (master—mirroredMaster, slave—mirroredSlave, or system—mirroredSystem).

c) **isAddressable** (mandatory) specifies the bus has addressing information. The **isAddressable** element is of type ***Boolean***. A value of ***True*** specifies these interfaces contain addressing information and a memory map can be traced through this interface. A value of ***False*** indicates these interfaces do not contain any traceable addressing information.

d) **extends** (optional) specifies if this definition is an extension from another bus definition. The **extends** element is of type ***libraryRefType*** (see X.Y.Z), it contains four attributes to specify a unique VLNV. See 6.12 on extending bus definitions.
   1) **vendor** attribute (mandatory) identifies the owner of the referenced description.
   2) **library** attribute (mandatory) identifies a library of referenced description.
   3) **name** attribute (mandatory) identifies a name of referenced description.
   4) **version** attribute (mandatory) identifies a version of referenced description.

e) **maxMasters** specifies the maximum number of masters that may appear on a bus. If the **maxMasters** element is not present, the numbers of masters is unbounded. The **maxMasters** elements is of type ***nonNegativeInteger***.

f) **maxSlaves** specifies the maximum number of slaves that may appear on a bus. If the **maxSlaves** element is not present, the numbers of slaves is unbounded. The **maxSlaves** elements is of type ***nonNegativeInteger***.

g) **systemGroupNames** (optional) defines an unbounded list of **systemGroupNames** elements, which in tern, define the possible group names to be used under an **onSystem** element in an abstraction definition. The definition of the group names in the bus definition allows multiple abstraction definitions to indicate which system interfaces match each other. The **systemGroupNames** element is of type ***Name***.

h) **description** (optional) allows a textual description of the interface. The type of this element is ***string***.

i) **vendorExtensions** (optional) contains any extra vendor-specific data related to the interface.

See also: SCR 9.1 and SCR 9.2.

6.2.3 Example

This is an example of an AHB **busDefinition**.

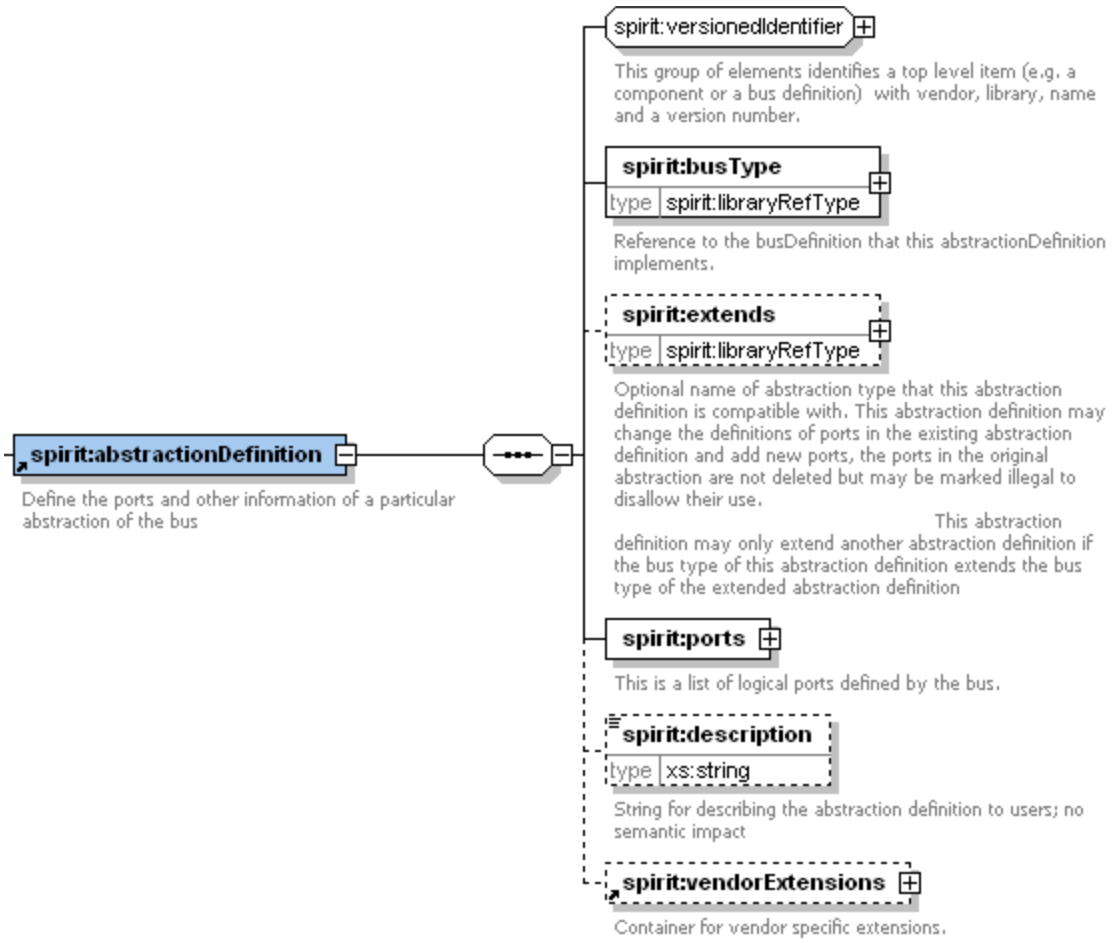
```
<?xml version="1.0" encoding="UTF-8" ?>
```

```
<spirit:busDefinition
xmlns:spirit= http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4
http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4/index.xsd">

    <spirit:vendor>amba.com</spirit:vendor>
    <spirit:library>AMBA</spirit:library>
    <spirit:name>AHB</spirit:name>
    <spirit:version>v1.0</spirit:version>
    <spirit:directConnection>false</spirit:directConnection>
    <spirit:isAddressable>true</spirit:isAddressable>
    <spirit:extends spirit:vendor="amba.com"
        spirit:library="AMBA"
        spirit:name="AHBlite"
        spirit:name=”v1.0” />
    <spirit:maxMasters>16</spirit:maxMasters>
    <spirit:maxSlaves>16</spirit:maxSlaves>
    <spirit:systemGroupNames>
        <spirit:systemGroupName>ahb_clk</spirit:systemGroupName>
        <spirit:systemGroupName>ahb_reset</spirit:systemGroupName>
    </spirit:systemGroupNames>
</spirit:busDefinition>
```

6.3 Abstraction definition

6.3.1 Schema

The following schema details the information contained in the **abstractionDefinition** element, which is one of the seven top-level elements in the IP-XACT specification used to describe the low-level aspects of a bus.



6.3.2 Description

The **abstractionDefinition** element describe the low-level aspects of a bus or interconnect. It contains the following elements and attributes.

a) The ***versionedIdentifier*** group provides a unique identifier; it consists of four subelements for a top-level IP-XACT element.

   1) **vendor** (mandatory) identifies the owner of this description. The recommended format of the **vendor** element is the company internet domain name.

   2) **library** (mandatory) identifies a library of this description. This allows one vendor to group descriptions.

   3) **name** (mandatory) identifies a name of this description.

   4) **version** (mandatory) identifies a version of this description. This allows one vendor to provide many descriptions which all have the same name, but are still uniquely identified.

b) **busType** (optional) specifies if this definition is an extension from another abstraction definition. The **busType** element is of type ***libraryRefType*** (see X.Y.Z), it contains four attributes to specify a unique VLNV. See 6.12 on extending bus definitions.

   1) **vendor** attribute (mandatory) identifies the owner of the referenced description.

   2) **library** attribute (mandatory) identifies a library of referenced description.

   3) **name** attribute (mandatory) identifies a name of referenced description.

   4) **version** attribute (mandatory) identifies a version of referenced description.

c) **extends** (optional) specifies if this definition is an extension from another abstraction definition. The **extends** element is of type ***libraryRefType*** (see X.Y.Z), it contains four attributes to specify a unique VLNV. See 6.12 on extending bus definitions.

   1) **vendor** attribute (mandatory) identifies the owner of the referenced description.
   2) **library** attribute (mandatory) identifies a library of referenced description.
   3) **name** attribute (mandatory) identifies a name of referenced description.
   4) **version** attribute (mandatory) identifies a version of referenced description.

d) **ports** (mandatory) is a list of logical ports, see 6.4.

e) **description** (optional) allows a textual description of the interface. The type of this element is ***string***.

f) **vendorExtensions** (optional) contains any extra vendor-specific data related to the interface.

The **abstractionDefinition** element contains a list of logical ports that define a representation of the bus type to which it refers. A port can be a wire port (see 6.7) or a transactional port (see 6.10). A *wire port* carries logic information or an array of logic information. A *transactional port* carries information that is represented on a higher level of abstraction.

An **abstractionDefinition** can extend another **abstractionDefinition** if and only if the bus type of the abstraction definition extends the bus type of the extending abstraction definition. The extending abstraction definition may change the definition of logical ports, add new ports, or mark existing logical ports illegal (to disallow their use).

See also: SCR 3.1, SCR 3.23, and SCR 3.24.

6.3.3 Example

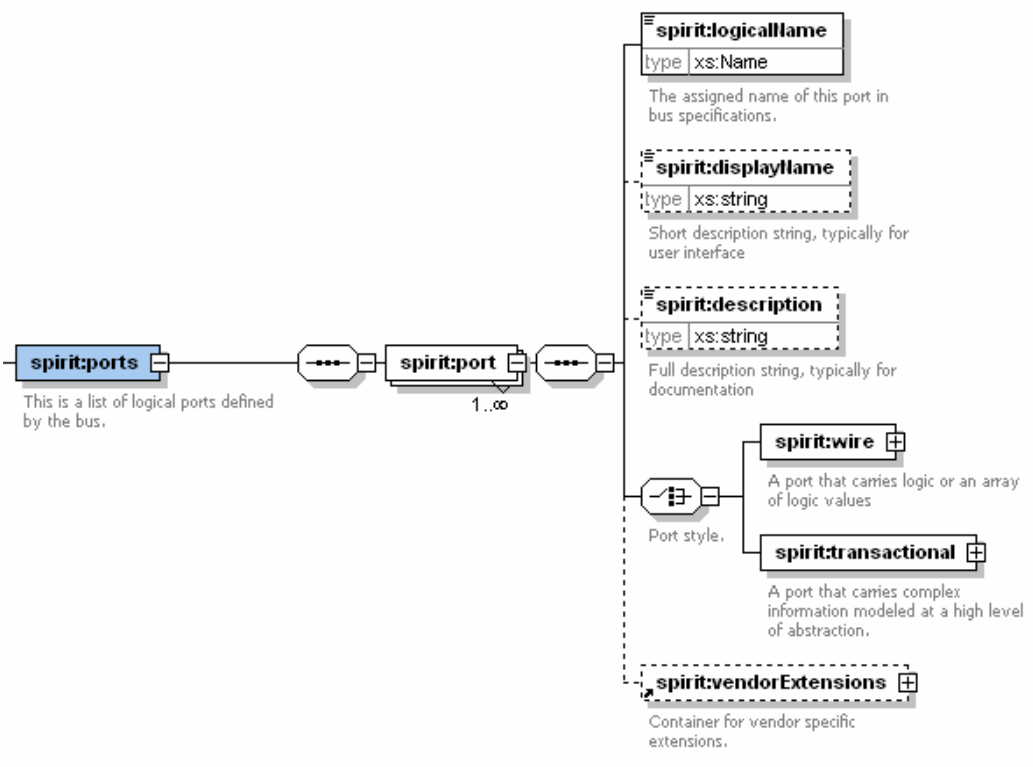
The following example shows an abstraction definition for the interrupt bus in the Leon2 TLM example.

```
<spirit:vendor>spiritconsortium.org</spirit:vendor>
<spirit:library>Leon</spirit:library>
<spirit:name>INT_PV</spirit:name>
<spirit:version>1.4</spirit:version>
<spirit:busType spirit:vendor="spiritconsortium.org"
spirit:library="Leon" spirit:name="Int" spirit:version="v1.0"/>
<spirit:ports>
   <spirit:port>
      <spirit:logicalName>INT_TRANSACTION</spirit:logicalName>
      <spirit:wire>
         <spirit:onMaster>
            <spirit:presence>required</spirit:presence>
            <spirit:direction>out</spirit:direction>
         </spirit:onMaster>
         <spirit:onSlave>
            <spirit:presence>required</spirit:presence>
            <spirit:direction>in</spirit:direction>
         </spirit:onSlave>
      </spirit:wire>
   </spirit:port>
</spirit:ports>
```

6.4 Ports

6.4.1 Schema

The following schema details the information contained in the **ports** element, which appears part of the **abstractionDefinition** element within an abstraction definition.



6.4.2 Description

The **ports** element is an unbounded list of **port** elements. Each **port** element defines the logical port information for the containing abstraction definition. It contains the following elements.

a) **logicalName** (mandatory) gives a name to the logical port that can be used later in component description when the mapping is done from a logical abstraction definition port to the components physical port. The type of this element is **Name**.

b) **displayName** (optional) allows a short descriptive text to be associated with the port. The type of this element is ***string***.

c) **description** (optional) allows a textual description of the port. The type of this element is ***string***.

d) Each **port** also requires a **wire** element or a **transactional** element to further describe the details about this port. See 6.5 or 6.10, respectively. A **wire** style port is a port that carries logic values or an array of logic values. A **transactional** style port is a port that carries any other type of information, typically used for TLM.

e) **vendorExtensions** (optional) contains any extra vendor-specific data related to the port.

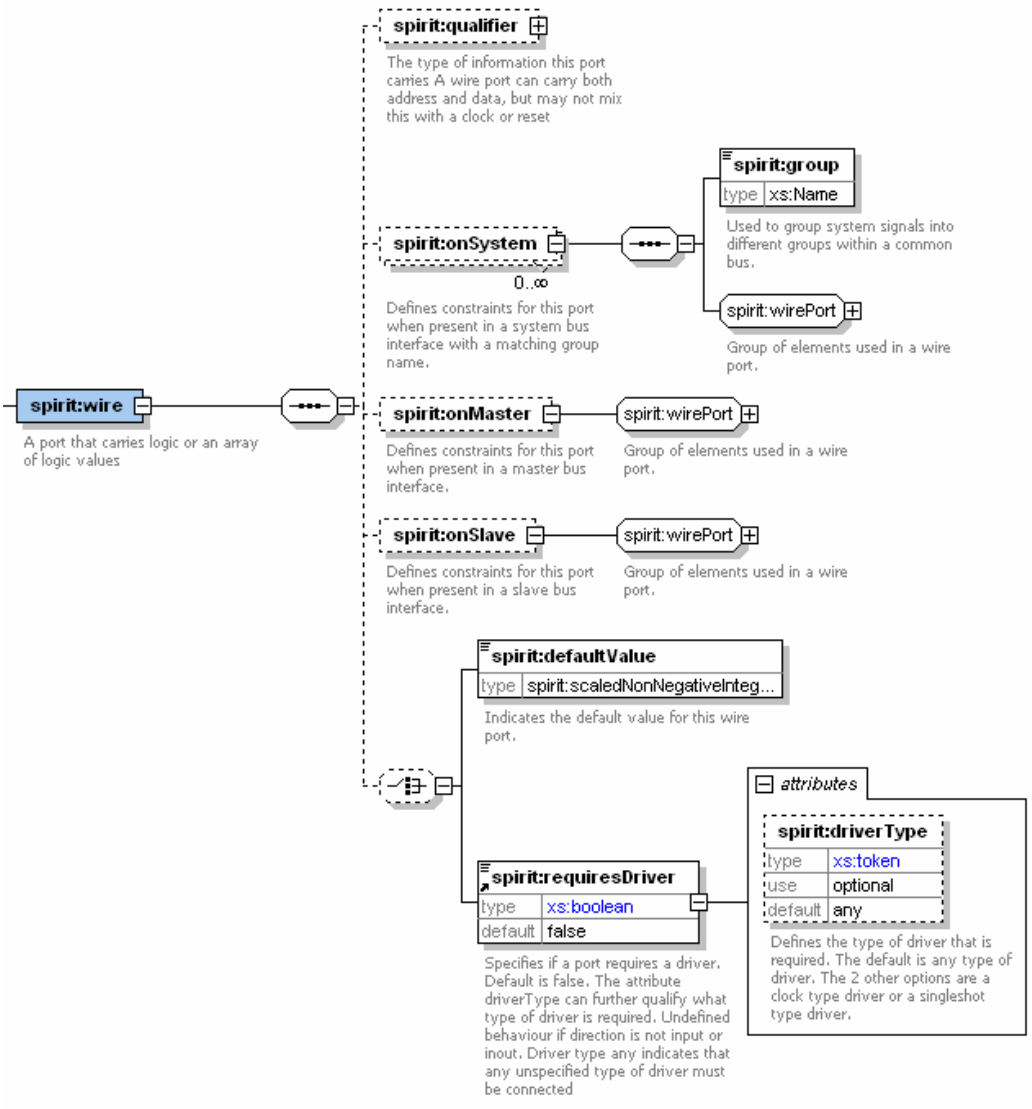
6.4.3 Example

See 6.3.3 for an example.

6.5 Wire ports

6.5.1 Schema

The following schema details the information contained in the **wire** element, which may appear as part of the **port** element within an abstraction definition (**abstractionDefinition**/**ports**/**port**).



6.5.2 Description

A **wire** element represents a port that carries logic values or an array of logic values. This logical wire port may provide optional constraints for a wire port, to which it is mapped inside a component or abstractor's **busInterface**. It contains the following elements and attributes.

a) **qualifier** (optional) indicates which type of information this wire port carries. See 6.6.

b) **onSystem** (optional) defines constraints, e.g., timing constraints, for this wire port if it is present in a system bus interface with a matching group name.

1) The **group** (mandatory) attribute indicates the group name for the wire port. It distinguishes between different sets of system interfaces. Usually, all the arbiter ports are processed together, or all the clock or reset ports are processed together. So, this is really a mechanism to specify any sort of non-standard bus interface capabilities for the interconnect. The **group** name shall match the one specified in the bus definition. The type of this element is ***Name***.

2) The group ***wirePort*** specifies what elements are used in this port. See 6.7.

c) **onMaster** (optional) defines constraints for this wire port when present in a master bus interface. The group ***wirePort*** specifies what elements are used in this port. See 6.7.

d) **onSlave** (optional) defines constraints for this wire port when present in a slave bus interface. The group ***wirePort*** specifies what elements are used in this port. See 6.7.

e) Either of the follow two element are allowed, but not both.

1) **defaultValue** (optional) contains the default logic value for this wire port. This value is applied when the port is left unconnected. The type of this element is ***scaledNonNegativeInteger***.

2) **requiresDriver** (optional) specifies whether the port requires a driver when used in a completed design. The type of this element is ***Boolean***. Its default value is ***False***, indicating this does not require a driver. When set to ***True***, the attribute **driverType** further qualifies what driver type is required: ***any*** (the default, meaning any logic signal or value), ***clock*** (meaning a repeating type waveform), or ***singleshot*** (a non-repeating type waveform).

NOTE—The **onMaster**, **onSlave**, and **onSystem** elements associated with each logical port provide optional constraints. So, if none of these constraints are specified, that port is unconstrained in how it appears in any interface. The abstraction definition author has the choice of how far to constrain the definitions. Generally speaking, more constraints in the definitions reduce implementation flexibility for whoever is creating bus IP that conforms to the abstraction definition.

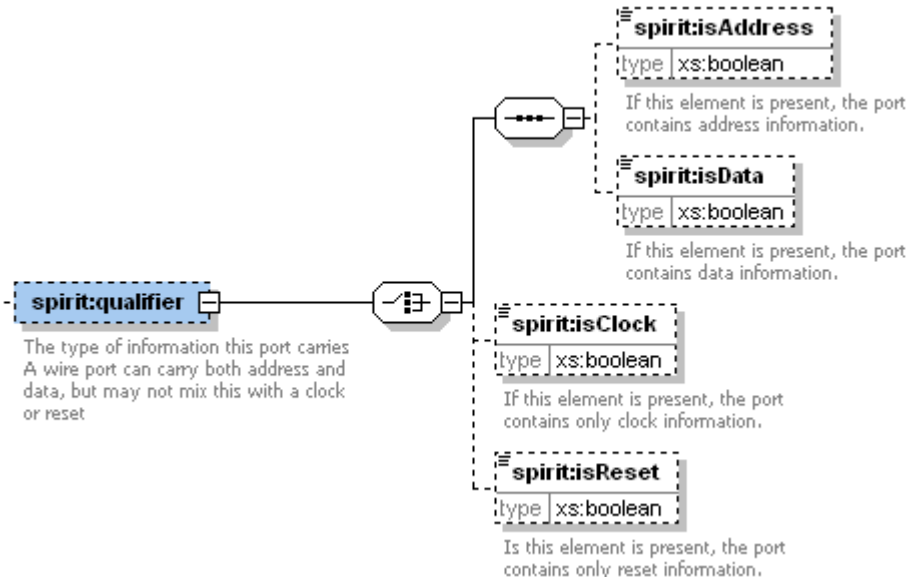
6.5.3 Example

See 6.3.3 for an example.

6.6 Qualifiers

6.6.1 Schema

The following schema details the information contained in the **qualifier** element, which may appear as part of the **wire** element within an abstraction definition (**abstractionDefinition**/**ports**/**port**/**wire**).



6.6.2 Description

The **qualifier** element indicates which type of information a wire port carries. It contains the following elements.

a) **isAddress** (optional), when ***True***, specifies the port contains address information. This **qualifier** may be paired with the **isData** element (useful for serial protocols). The type of this element is ***Boolean***.

b) **isData** (optional), when ***True***, specifies the port contains data information. This data resides in registers defined in the memory map referenced by the interface. The width defined by the port on each side of the two connected bus interfaces can be used to determine which portions of the data may be lost or gained (tied off to defaults) during transfers if the two widths do not match. This **qualifier** may be paired with the **isAddress** element (useful for serial protocols). The type of this element is ***Boolean***.

c) **isClock** (optional), when ***True***, specifies this signal is a clock for this bus interface, i.e., it provides a repeating signal which the interface uses to implement the protocol. No method of processing is implied with this tag. This tag shall only be applied to pure clock signals. This **qualifier** may not be combined with other qualifiers. The type of this element is ***Boolean***.

d) **isReset** (optional), when ***True***, specifies this signal is a reset for this bus interface., i.e., it provides the necessary input to put the interface into a known state. No method of processing is implied with this tag. This tag should only be applied to pure reset signals. This **qualifier** may not be combined with other qualifiers. The type of this element is ***Boolean***.

See also: SCR 9.1 and SCR 9.2.

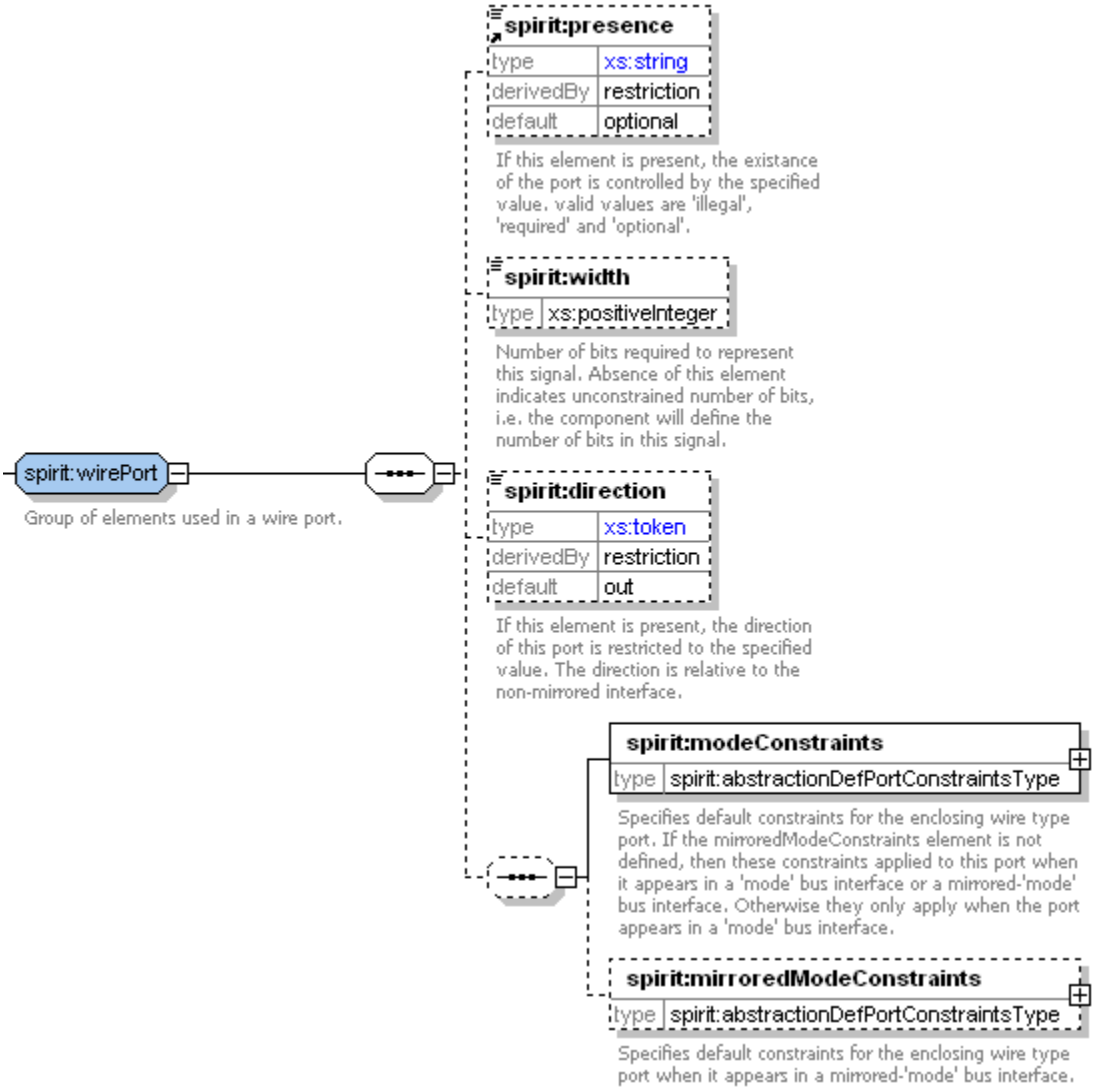
6.6.3 Example

See 6.3.3 for an example.

6.7 Wire port group

6.7.1 Schema

The following schema details the information contained in the ***wirePort*** group, which may appear as part of the **onSystem**, **onMaster**, or **onSlave** element within a **wire** element within an abstraction definition (**abstractionDefinition**/**ports**/**port**/**wire**/**on*mode***).



6.7.2 Description

The group ***wirePort*** specifies what elements are used in a **wire** port. It contains the following elements.

a) **presence** (optional) provides the capability to require or forbid a port from appearing in a **busInterface**. The three possible values are ***illegal***, ***required***, or ***optional*** (the default).

b) **width** (optional) represents the number of logical bits that are required to represent this signal. When mapping to this logical port in a **busInterface/portmap**, the numbering shall start from `0` to `width-1`. If **width** is not specified, the component shall define the number of bits in this signal, but

the logical portmap numbering shall still start at 0. If necessary, logical bit 0 shall be the least significant bit. The **width** element is of type ***positveInteger***.

c) **direction** (optional) restricts the direction of the port relative to the non-mirrored interface. The three possible values are ***in***, ***out*** (the default), or ***inout***.

d) Each ***wirePort*** group can also have a sequence of **modeConstraints** and **mirroredModeConstraints** specifying the default constraints of this interface during synthesis. The **modeConstraints** apply to this port if it appears in a non-mirrored 'mode' bus interface (see 6.8). Any **mirroredModeConstraints** apply to this port if it appears in a mirrored-'mode' bus interface (see 6.9).

If **mirroredModeConstraints** are not specified, the **modeConstraints** also apply to this port in a mirrored-'mode' bus interface.

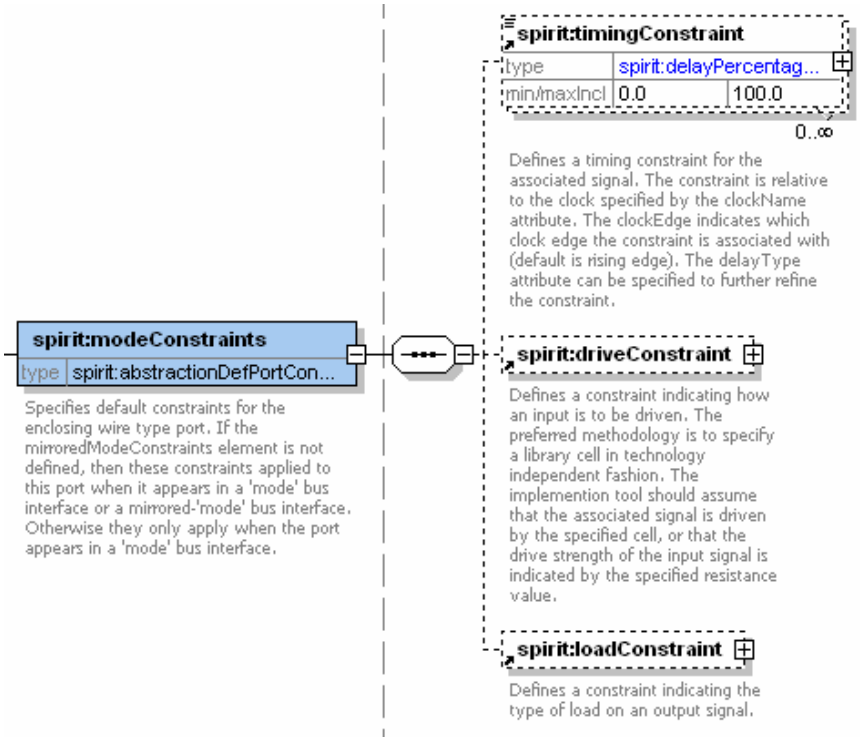
6.7.3 Example

See 6.3.3 for an example.

6.8 Wire port 'mode' constraints

6.8.1 Schema

The following schema defines the information contained in the **modeConstraints** element, which may appear within an **onMaster**, **onSlave**, or **onSystem** element within an abstraction definition (**abstractionDefinition/ports/port/wire/on*mode***).



6.8.2 Description

The **modeConstraints** element defines any default implementation constraints associated with the containing wire port of the abstraction definition. It contains the following elements.

a) **timingConstraint** (optional) element defines a technology-independent timing constraint associated with the containing wire port. See 7.11.13.

b) **driveConstraint** (optional) element defines a technology-independent drive constraint associated with the containing wire port. See 7.11.12.

c) **loadConstraint** (optional) element defines a technology-independent load constraint associated with the containing wire port. See 7.11.11.

The constraints contained within the **modeConstraints** element are only applied to the corresponding physical port in a component when the physical port does not have any constraints defined within its own port element and there is no SDC file associated with the component. For example, if it appears inside an **onMaster** element, the constraints apply when the port appears in a master interface. If the **modeConstraints** element is immediately followed by a **mirroredModeConstraints** element (see 6.9), the constraints defined in the **modeConstraints** element apply only when the port is used in a non-mirrored *mode* interface. Otherwise, the constraints apply when the port appears in a *mode* interface or a mirrored-*mode* interface.

6.8.3 Example

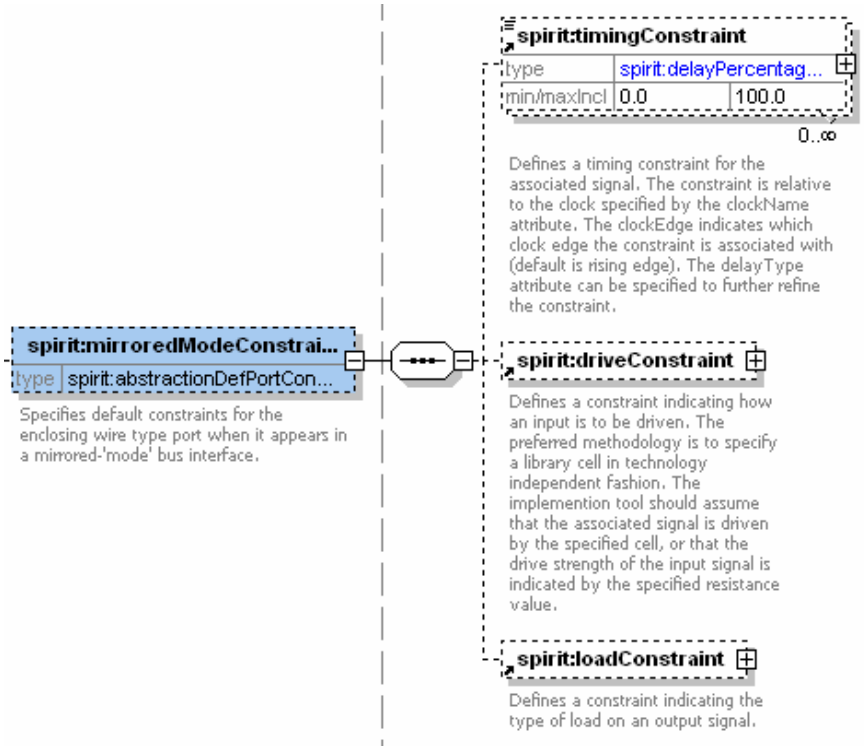
The following example shows a port within an abstraction definition, containing a single timing constraint. Since there is no `mirroredModeConstraint` element, this timing constraint applies when the `HRDATA` port appears in either a master interface or a mirrored-master interface.

```
<spirit:port>
   <spirit:logicalName>HRDATA</spirit:logicalName>
   <spirit:wire>
      <spirit:onMaster>
         <spirit:modeConstraints>
            <spirit:timingConstraint spirit:clockName="HCLK">40
               </spirit:timingConstraint>
         </spirit:modeConstraints>
      </spirit:onMaster>
   </spirit:wire>
</spirit:port>
```

6.9 Wire port mirrored-'mode' constraints

6.9.1 Schema

The following schema defines the information contained in the **mirroredModeConstraints** element, which may appear within an **onMaster**, **onSlave**, or **onSystem** element within an abstraction definition (**abstractionDefinition**/**ports**/**port**/**wire**/**on*mode***).



6.9.2 Description

The **mirroredModeConstraints** element also defines any default implementation constraints associated with the containing wire port of the abstraction definition. It contains the following (optional) elements.

a) **timingConstraint** (optional) element defines a technology-independent timing constraint associated with the containing wire port. See 7.11.13.

b) **driveConstraint** (optional) element defines a technology-independent drive constraint associated with the containing wire port. See 7.11.12.

c) **loadConstraint** (optional) element defines a technology-independent load constraint associated with the containing wire port. See 7.11.11.

The constraints contained within the **mirroredModeConstraints** element are only applied to the corresponding physical port in a component when the physical port does not have any constraints defined within its own port element and there is no SDC file associated with the component. For example, if it appears inside an **onMaster** element, the constraints only apply when the port appears in a mirrored-master interface.

6.9.3 Example

The following example shows a port within an abstraction definition, containing a single timing constraint. On a master interface the port gets 40% of the cycle time and on a mirrored master interface it gets 60% of the cycle time.

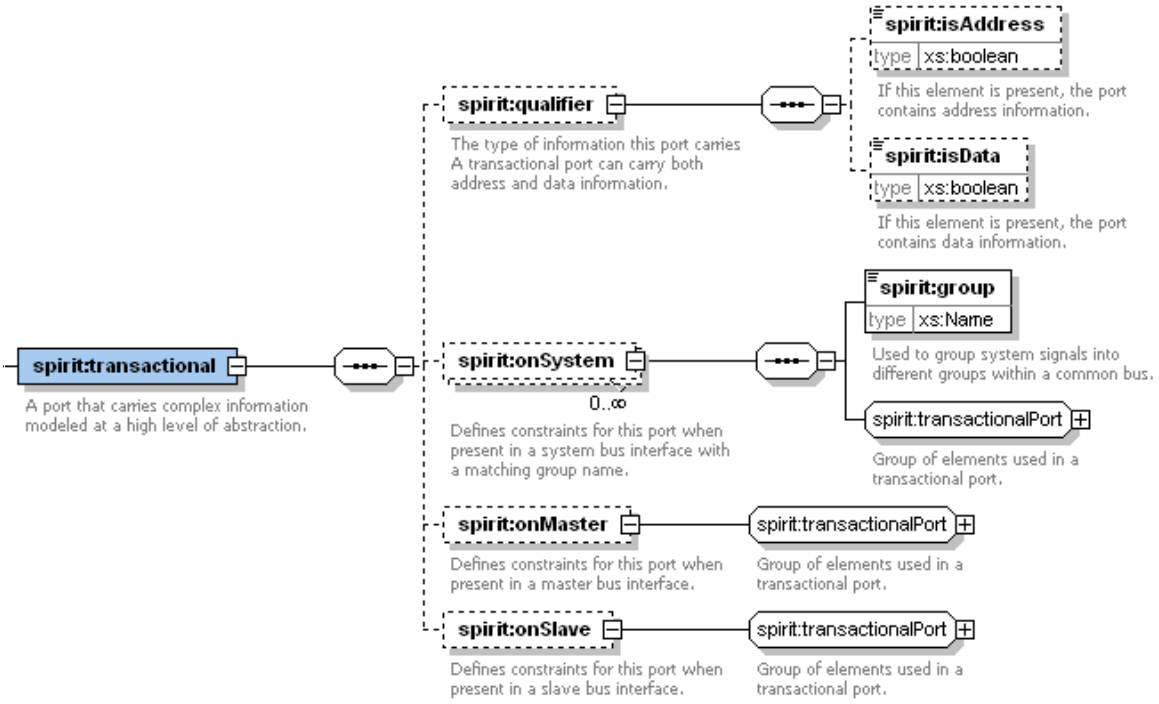
```
<spirit:port>
   <spirit:logicalName>HRDATA</spirit:logicalName>
   <spirit:wire>
      <spirit:onMaster>
```

```
            <spirit:modeConstraints>
                <spirit:timingConstraint spirit:clockName="HCLK">40
                    </spirit:timingConstraint>
            </spirit:modeConstraints>
            <spirit:mirroredModeConstraints>
                <spirit:timingConstraint spirit:clockName="HCLK">60
                    </spirit:timingConstraint>
            </spirit:mirroredModeConstraints>
        </spirit:onMaster>
    /spirit:wire>
</spirit:port>
```

6.10 Transactional ports

6.10.1 Schema

The following schema defines the information contained in the **transactional** element, which may appear within a **port** within an abstraction definition (**abstractionDefinition**/**ports**/**port**).



6.10.2 Description

The **transactional** element defines a logical transactional port of the abstraction definition. This logical transactional port may provide optional constraints for a transactional port, to which it is mapped inside a component or abstractor's **busInterface**. The **transactional** element also contains the following elements and attributes.

a) The **qualifier** (optional) element indicates which type of information this transactional port carries. It contains either or both of the following elements.

   1) **isAddress** (optional) specifies the port contains address information.

   2) **isData** (optional) specifies the port contains data information.

b) **onSystem** defines constraints for this transactional port if it is present in a system bus interface with a matching group name.

   1) The **group** attribute indicates the group name for the transactional port. It distinguishes between different sets of system interfaces. Usually, all the arbiter ports are processed together, or all the clock or reset ports are processed together. So, this is really a mechanism to specify any sort of non-standard bus interface capabilities for the interconnect. The **group** name shall match the one specified in the bus definition.

   2) The group ***transactionalPort*** specifies what elements are used in this port. See 6.11.

c) **onMaster** defines constraints for this transactional port when present in a master bus interface. The group ***transactionalPort*** specifies what elements are used in this port. See 6.11.

d) **onSlave** defines constraints for this transactional port when present in a slave bus interface. The group ***transactionalPort*** specifies what elements are used in this port. See 6.11.

See also: SCR 6.14 and SCR 6.17.

6.10.3 Example

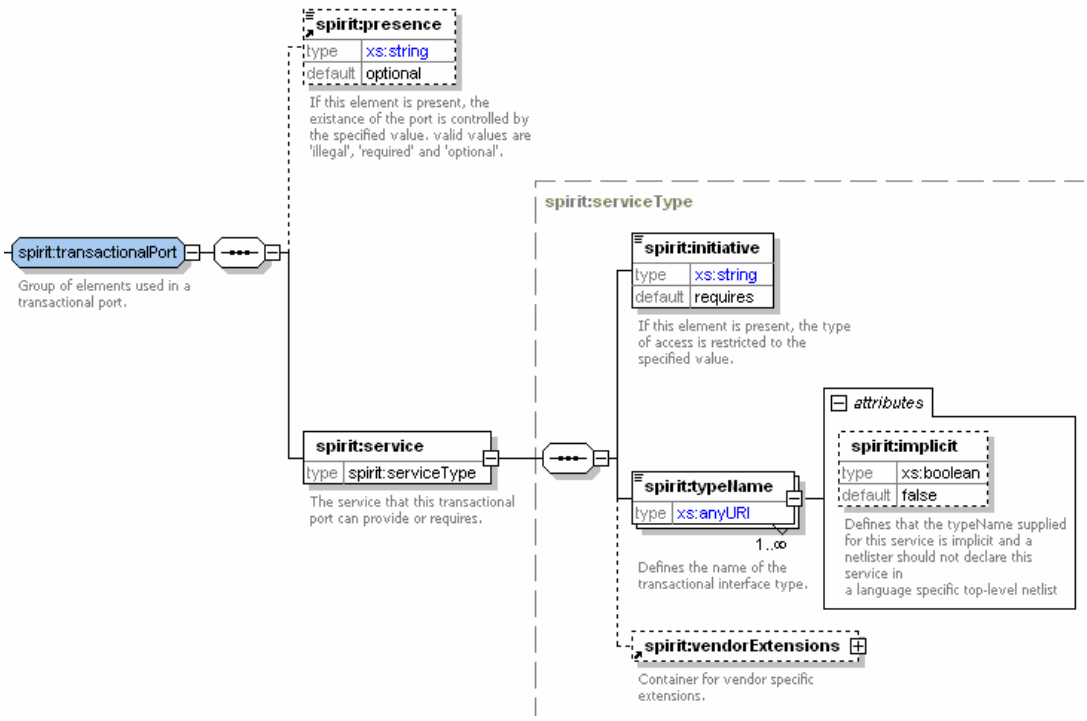
The following example shows a transactional port within an abstraction definition, carrying data information.

```
<spirit:port>
   <spirit:logicalName>pv_data</spirit:logicalName>
   <spirit:transactional>
      <spirit:qualifier>
         <spirit:isData>true</spirit:isData>
      </spirit:qualifier>
      <spirit:onMaster>
         <spirit:presence>required</spirit:presence>
         <spirit:service>
            spirit:initiative>requires</spirit:initiative>
            <spirit:typeName>pv_basic_type</spirit:typeName>
         </spirit:service>
      </spirit:onMaster>
   </spirit:transactional>
</spirit:port>
```

6.11 Transactional port group

6.11.1 Schema

The following schema defines the information contained in the **transactionalPort** group, which may appear within an **onMaster**, **onSlave**, or **onSystem** element within an abstraction definition (**abstractionDefinition**/**ports**/**port**/**transactional**/**on*mode***).



6.11.2 Description

A **transactionalPort** group contains elements defining constraints associated with a transactional logical port within an **abstractionDefinition**. It contains the following elements.

a) **presence** (optional) provides the capability to require or forbid a port to appear in a **busInterface**. Its three possible values are ***illegal***, ***required***, or ***optional*** (the default).

b) **service** (mandatory) defines constraints on the service type, which the component transactional port can provide or require. It also contains the following elements or attributes.

   1) **initiative** (mandatory) defines the type of access: ***requires*** (the default), ***provides***, or ***both***. For example, a SystemC `sc_port` is defined using **requires**, since it requires a SystemC interface.

   2) **typeName** (mandatory) is and unbounded list that defines the names of the transactional interface types. The typeName element is of type ***anyURI***. The **implicit** (optional) attribute may be be used here to indicate this element is implicit and a netlister shall not declare this service in a language-specific top-level netlist.

   3) **vendorExtensions** contains any extra vendor-specific data related to the interface.

See also: SCR 6.5.1, SCR 6.5.2, SCR 6.5.3, and SCR 6.7.

6.11.3 Example

The following example shows a custom transactional port within an abstraction definition. Constraints are defined for transactional port used in `master` or `slave` interfaces.

```
<spirit:port>
   <spirit:logicalName>custom_tlm_port</spirit:logicalName>
   <spirit:transactional>
```

```
        <spirit:onMaster>
            <spirit:service>
                <spirit:initiative>provides</spirit:initiative>
                <spirit:typeName implicit="true">TLM
                    </spirit:typeName>
            </spirit:service>
        </spirit:onMaster>
        <spirit:onSlave>
            <spirit:service>
                <spirit:initiative>requires</spirit:initiative>
                <spirit:typeName implicit="true">TLM
                    </spirit:typeName>
            </spirit:service>
        </spirit:onSlave>
    </spirit:transactional>
</spirit:port>
```

6.12 Extending bus and abstraction definitions

6.12.1 Extending bus definitions

Bus definitions may use the **extends** element to create a family of compatible inter-connectable bus definitions. A bus definition (`B`) extends another existing bus definition (`A`) by specifying the **extends** element in the `B` bus definition's element list. Bus definition `B` is referred to as the *extending* bus definition and bus definition `A` is referred to as the *extended* bus definition. For two bus definitions related by the **extends** relation to be inter-connectable, they need to be in a direct line of descent in the hierarchical *extension* tree, as illustrated in Figure 7.

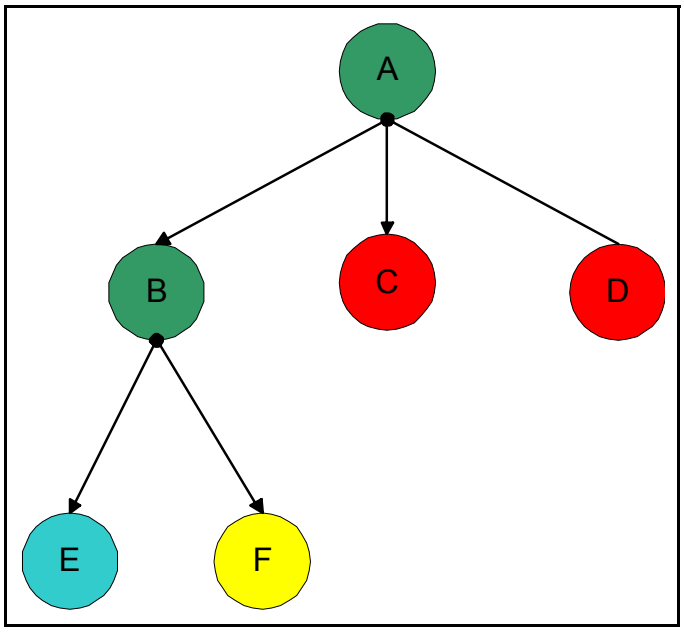


Figure 7—Extends relation hierarchy tree

In Figure 7, bus definition `B` extends bus definition `A`. Bus interfaces of bus definition `E` shall only be connected with bus interfaces of bus definitions `E`, `B`, and `A`. By the same token, bus interfaces of bus definition `F` shall only be connected with bus interfaces of bus definitions `F`, `B`, and `A`.

6.12.2 Extending abstraction definitions

The **abstractionDefinition** that references the *extended* **busDefinition** via the **busType** element is referred to as the *extended* **abstractionDefinition**. The bus definition writer shall supply an **abstractionDefinition** that references the *extending* **busDefinition** and it is referred to as the *extending* **abstractionDefinition**. The *extending* **abstractionDefinition** shall reference the *extended* **abstractionDefinition** via its **extends** element. An example of extending is shown in Figure 8.

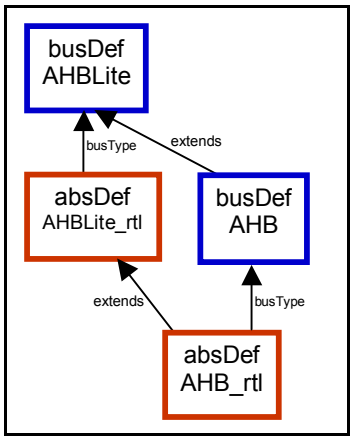


Figure 8—Example of extending

The *extending* bus definition and abstraction definition pair shall be able to stand on its own independent of the *extended* bus definition and abstraction definition pair; therefore, all the elements and attributes of the *extended* bus definition and abstraction definition pair shall be specified in the *extending* bus definition and abstraction definition pair. Also, all the ports in the *extended* abstraction definition shall be explicitly defined in the *extending* abstraction definition. Some of the elements and attributes of the *extending* bus definition and abstraction definition pair may be modified from the *extended* bus definition and abstraction definition pair, while others may not.

6.12.3 Modifying definitions

Table 1 specifies which elements and attributes may be modified in a bus definition and Table 2 specifies which elements and attributes may be modified in an abstraction definition.

Table 1—Elements of *extending* bus definition

Item	Modified	Comment
directConnection	No	
isAddressable	No	
maxMasters	Yes	Smaller number applies
maxSlaves	Yes	Smaller number applies
systemGroupNames	Yes	New group names may be added
description	Yes	
vendorExtensions	Yes	

Table 2—Elements of *extending* abstraction definition

Item	Modified	Comment
ports	Yes	See Table 3
description	Yes	
vendorExtensions	Yes	

The *extending* abstraction definition may add new ports and the *extending* abstraction definition may mark certain ports as illegal to disallow their use. Table 3 specifies which port elements may be modified when extending bus definitions.

Table 3—Elements of a port in an *extending* abstraction definition

Item	Modified	Comment
logicalName	No	Changing this name implies a port that is different than the one in the *extended* **abstractionDefinition**.
requiresDriver	Yes	
isAddress	No	
isData	No	
isClock	No	
isReset	No	
onSystem/group	Yes	
presence	Yes	
width	Yes	
direction	No	
modeConstraints	Yes	
mirroredModeConstraints	Yes	
defaultValue	Yes	
service/initiative	No	
service/typeName	No	
service/vendorExtensions	Yes	
vendorExtensions	Yes	

6.12.4 Interface connections

When a bus interface of the *extended* bus definition and abstraction definition pair is connected with a bus interface of the *extending* bus definition and abstraction definition pair, it is possible either interface may have unconnected ports due to the previous extensions of the port list (i.e., port additions or disownment).

The bus definition writer needs to be aware of these scenarios and specify **defaultValue**s where necessary. Here is a sample of the possible connections between two extended interfaces (`A` and `B`).

master(`A`) connecting to slave(`B`) (if **directConnection** = *True*)

master(`A`) connecting to mirror-master(`B`)

slave(`A`) connecting to mirror-slave(`B`)

master(`B`) connecting to slave(`A`) (if **directConnection** = *True*)

master(`B`) connecting to mirror-master(`A`)

slave(`B`) connecting to mirror-slave(`A`)

6.13 Clock and reset handling

GEE--This needs a new home

Abstraction definitions shall include all the logical ports that can participate in the protocol of the bus and bus interfaces need to map to the component all the logical ports that participate in the protocol of that bus at that interface. For example, on an AXI bus, the ports of the write channel can participate in the protocol of the bus, so they shall be included in the AXI abstraction definition. These ports will participate in the protocol at any AXI bus interface that supports writes, so they need to be included in all such bus interfaces, but not included in any AXI bus interfaces that only support reads.

This requirement applies to clock and ports signals as much as it does to other ports. If the protocol of a bus is dependent on a clock or reset port, the bus definition for that bus shall include that clock or reset port. Similarly if the bus protocol at a bus interface is dependent on a particular clock or reset port, the port map of that bus interface shall include that port. The clock or reset port, however, do not need to exist as a port of the component implementation, since it may be mapped to a phantom port of the component (see 7.11.16.3.2). Also, since multiple bus ports may be mapped to a single component port (and component ports may also participate in ad-hoc connections), the clock routing is not required to match or be defined by the bus infrastructure.

In some cases, a component may have clock or reset ports that are not associated with and do not participate in the protocol of any bus interface, but do provide a clock or reset signal to the internal logic of the component instead, e.g., a processor clock. In such cases, the clock port should be included in a special purpose clock or reset bus interface, with an appropriate special purpose bus type, or not be mapped into any interface and connected using ad-hoc connections instead.

7. Component descriptions

7.1 Components

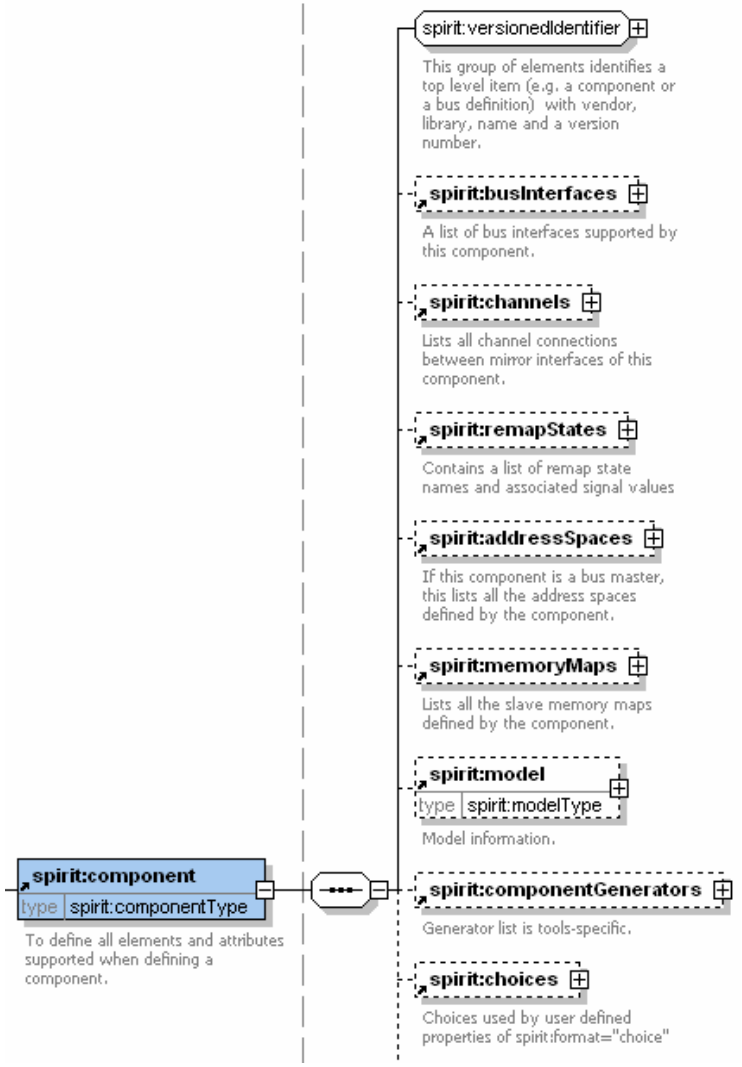
An IP-XACT *component* is the central placeholder for the objects meta-data. Components are used to describe cores (processors, co-processors, DSPs, etc.), peripherals (memories, DMA controllers, timers, UART, etc.), and buses (simple buses, multi-layer buses, cross bars, network on chip, etc.). An IP-XACT component can be of two kinds: static or configurable. A DE cannot change a *static component*. A *configurable component* has configurable elements (such as parameters) that can be configured by the DE and these elements may also configure the RTL or TLM model.

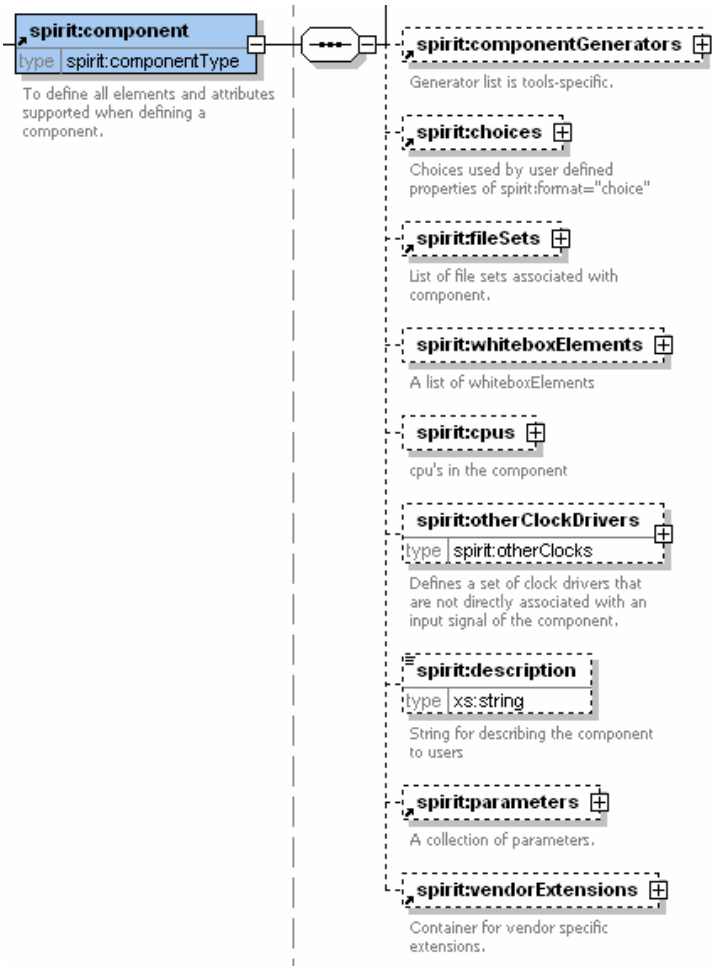
An IP-XACT component can be a hierarchical object or a leaf object. *Leaf components* do not contain other IP-XACT components, while *hierarchical components* contain other IP-XACT sub-components. This can be recursive by having hierarchical components that contain hierarchical components, etc.—leading to the concept of *hierarchy depth*. The IP being described may have a completely different hierarchical arrangement in terms of its implementation in RTL or TLM to that of its IP-XACT description. So, a description of a large IP component may be made up of many levels of hierarchy, but its IP-XACT description need only be a leaf object as that completely describes the IP. On the other hand, some IP can only be described in terms of a hierarchical IP-XACT description, no matter what the arrangement of the implementation hierarchy.

An IP-XACT component may contain a channel or a bridge. A *channel* is a special IP-XACT object that can be used to describe multi-point connections between regular components that may require some interface adaptation. A *bridge* is a point-to-point reference of slave to master interfaces. Both of these concepts are used to describe the interconnect between components.

7.1.1 Schema

The following schema details the information contained in the **component** element, which is one of the seven top-level elements in the IP-XACT specification used to describe a component.





7.1.2 Description

Each element of a **component** is detailed in the rest of this clause; the main sections of a **component** are:

a) ***versionedIdentifier*** group provides a unique identifier; it consists of four subelements for a top-level IP-XACT element.

   1) **vendor** (mandatory) identifies the owner of this description. The recommended format of the **vendor** element is the company internet domain name.

   2) **library** (mandatory) identifies a library of this description. This allows one vendor to group descriptions.

   3) **name** (mandatory) identifies a name of this description.

   4) **version** (mandatory) identifies a version of this description. This allows one vendor to provide many descriptions which all have the same name, but are still uniquely identified.

b) **busInterfaces** (optional) specifies all the interfaces for this component. A **busInterface** is a grouping of ports related to a function, typically a bus, defined by a bus definition and abstraction definition. See 7.5.

c) **channels** (optional) specifies the interconnection between interfaces inside of the component. See 7.6.

d) **remapStates** (optional) specifies the combination of logic states on the component ports and translates them into a logical name for use by logic that controls the defined address map. See 7.9.2.

e) **addressSpaces** (optional) specifies the addressable area as seen from a **busInterface** with an interface mode of **master**. See 7.7.

f) **memoryMaps** (optional) specifies the addressable area as seen from a **busInterface** with an interface mode of **slave**. See 7.8.

g) **model** (optional) specifies all the different views, ports, and model configuration parameters of the component. See 7.11.

h) **componentGenerators** (optional) specifies a list of generator programs attached to this component. See 7.12.

i) **choices** (optional) specifies multiple enumerated lists. These lists are referenced by other sections of this component description. See 7.14.

j) **fileSets** (optional) specifies groups of files and possibly their function for reference by other sections of this component description. See 7.13.

k) **whiteboxElements** (optional) specifies all the different locations in the component that can be accessed for verification purposes. See 7.15.

l) **cpus** (optional) indicates this component contains programmable processors. See 7.17.

m) **otherClockDrivers** (optional) specifies any clock signals, which are not external ports on the component, where implementation constraints are associated. See 7.11.15.

n) **description** (optional) allows a textual description of the component. The **description** element is of type ***string***.

o) **parameters** (optional) describes any **parameter** that can be used to configure or hold information related to this component. See X.Y.Z.

p) **vendorExtensions** (optional) contains any extra vendor-specific data related to the component. See X.Y.Z.

7.1.3 Example

GEE--This example needs to be filled out more. Maybe just reference a large example at the end ....

This is an example of a component (a Leon Timer peripheral).

```
<?xml version="1.0" encoding="UTF-8" ?>

<spirit:component
    xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4
    http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4/index.xsd">
    <spirit:vendor>spiritconsortium.org</spirit:vendor>
    <spirit:library>Leon2</spirit:library>
    <spirit:name>timers</spirit:name>
    <spirit:version>1.00</spirit:version>
    <spirit:busInterfaces>
    ...
    <spirit:memoryMaps>

    <spirit:model>
    ...
    <spirit:choices>
    ...
    <spirit:fileSets>
    ...
```

```
</spirit:component>
```

7.2 Interfaces

Each IP component normally identifies one or more bus interfaces. *Bus interfaces* are groups of ports that belong to an identified bus type (i.e., a reference to a **busDefinition** (see 6.2)) and an abstraction type (i.e., a reference to an **abstractionDefinition** (see 6.3)). The purpose of the bus interface is to map the physical signals of the component to the logical ports of the abstraction definition. This mapping provides more information about the interface.

There are seven possible modes for a bus interface: master, slave, and system; each with two flavors: direct and mirrored. Additionally, a monitor interface can be used to connect IP into the design for verification.

7.2.1 Direct interface modes

A *master interface* is the interface mode that initiates a transaction (like a read or write) on a bus. Master interfaces tend to have *associated address spaces* (address spaces with programmers view).

A *slave interface* is the interface mode that terminates or consumes a transaction initiated by a master interface. Slave interfaces often contain information about the registers that are accessible through the slave interface.

A *system interface* is neither a master nor slave interface; this interface mode allows specialized (or non-standard) connections to a bus, such as external arbiters. System interfaces can be used to handle situations not covered by the bus specification or deviations from the bus specification standard.

The following guidelines also apply to the direct interface modes.

— If a port's functionality is documented in the bus's documentation, then it shall be included in master and slave interfaces; only those ports that do not have documented functionality should be included in system interfaces.

— Some buses have specialized sideband ports. If these are tied or related to the standard ports in the bus (as opposed to being completely standalone), these ports should have some sort of **system** element designator in the bus definition.

7.2.2 Mirrored interface modes

As the name suggests, a *mirrored interface* has the same (or similar) ports to its related direct bus interface, but each port's direction or initiative is reversed. So a port that is an input on a direct bus interface would be an output in the matching mirrored interface. A mirrored bus interface (like its non-mirrored counterpart) supports the master, slave, and system classes.

7.2.3 Monitor interface modes

A *monitor interface* connects verification IP used to a master, slave, system, mirrored-master, mirrored-slave, or mirrored-system for observation. The connection shall not modify the connected interfaces. A monitor interface is identified by using the **monitor** element in the interface definition and specifying the type of active interface being monitored (master, slave, etc.).

7.3 Interface interconnections

IP-XACT provide for three different types of connections between interfaces. A *direct connection* is a connection between a master interface and a slave interface. A *direct-mirrored connection* is a connection between a direct interface and its corresponding mirrored interface (i.e. slave and mirrored-slave). A *monitor connection* is a connection between any interface type (other than monitor) and a monitor interface. It is not possible to connect two mirrored interfaces.

All interconnections are described in a top-level design object. See X.Y.Z.

7.3.1 Direct connection

A direct connection is a connection between a master interface and a slave interface. This connection is a single point-to-point connection. More complex connection schemes with direct connections are possible with the use of a **bridge** component. The direct connection shall meet the following conditions and rules.

a) The bus definition permits a direct connection, as specified in the bus definition. See 6.2.

b) The two interfaces shall be of the same or extended bus definitions and/or extended abstraction definitions.

c) For addressable buses:

1) The value of **bitsInLau** at the master and the slave shall match.

2) The value of **endianness** at the master and the slave shall match.

3) The value of **bitSteering** at the master and the slave shall match.

4) The address range defined on the slave interface shall be less than or equal to the address range defined on the master interface.

7.3.2 Direct-mirrored connection

A direct-mirrored connection is a connection between a master interface and a mirrored-master interface, a slave interface and a mirrored-slave interface, or a system interface and a mirrored-system interface. These connections are all single point-to-point connections. More complex connection schemes with direct-mirrored connections are possible with the use of a **channel** component. The direct-mirrored connection shall meet the following rules.

a) The two interfaces shall be of the same or extended bus definitions and/or extended abstraction definitions.

b) For addressable buses:

1) The value of **bitsInLau** at the master and the slave shall match.

2) The value of **endianness** at the master and the slave shall match.

3) The value of **bitSteering** at the master and the slave shall match.

7.3.3 Monitor connection

A monitor connection is a connection between a monitor interface and any other interface mode, master, mirrored-master, slave, mirrored-slave, system, or mirrored-system interface. The monitor interface is defined for only one mode and can only be used with that specific mode. Monitor connections are purely for non-intrusive observation of an interface. These connections are single-point to multi-point connections: the single point being the interface to be monitored and the multi-point being the monitor interface. More than one monitor may be attached to the same interface. The monitor connection shall meet the following rules.

a) The monitor interface mode shall match the monitored interface mode.

b) The two interfaces shall be of the same or extended bus definitions and/or extended abstraction definitions.

c) The connection of a monitor interface shall not count as a connected interface in the determination of the maximum master or maximum slave calculations.

7.3.4 Interface logical to physical port mapping

An interface on a component contains a port map to associate the physical ports on the component with the logical ports in the abstraction definition. This mapping is what provides the extra information needed to enable higher level of design.

A *physical port* defined in a component is assigned a physical port name and optionally can be assigned a **left** and a **right** element to represent a vector. The **left** element indicates the first boundary, the **right** element, the second boundary. **left** may be larger than **right** and that **left** may be the MSB or LSB (the **right** being the opposite). The **left** and **right** elements are the (bit) rank of the left-most and right-most bits of the port.

A *logical port* defined in an abstraction definition is assigned a logical port name and, optionally, a width. The logical port is assigned a numbering from `0` to the `width-1` if the width is present. If the width is not present, the logical port number shall start at `0` and not have an upper bound.

7.3.4.1 Mapping rules

These rules describe the assignment of logical bit numbers to a physical port.

a) If both ports have a vector defined, the logical port width=max(logical.left,logical.right) - min(logical.left,logical.right) +1 shall be equal to the physical port width=max(physical.left,physical.right) - min(physical.left,physical.right) +1. The mapping is such that logical.left-> physical.left down to logical.right-> physical.right.

b) If only the physical port has a vector defined, the logical port width=(width from abstraction definition, if defined) shall be equal to the physical port width=max(physical.left,physical.right) - min(physical.left,physical.right) +1. The mapping is such that logical.width-1-> physical.left down to logical.0-> physical.right.

c) If only the logical port has a vector defined, then logical port width=max(logical.left,logical.right) - min(logical.left,logical.right) +1 shall be equal to the physical port width=max(port.left,port.right) - min(port.left,port.right) +1. The mapping is such that logical.left-> port.left down to logical.right-> port.right.

d) If neither vector is defined, the logical port width=(width from abstraction definition, if defined) shall be equal to the physical port width=max(port.left,port.right) - min(port.left,port.right) +1. The mapping is such that logical.width-1-> port.left down to logical.0-> port.right.

7.3.4.2 Physical interconnections

With all logical bits having been assigned from the abstraction definition to physical port, it is a simple matter to describe the physical connections that result from an interface connection. All connections are made purely based on the logical bit assignment. Like logical bit numbers from each interface are connected. The alignment is always such that logical bit `0` from interface `A` connects to logical bit `0` from interface `B`, logical bit `1` from interface `A` connects to logical bit `1` from interface `B`, and so on.

7.4 Complex interface interconnections

There are two constructs used to connect interfaces of standard components together (traditional components, usually with ‘masters’ and ‘slave’ interfaces), a channel and a bridge. These constructs are also encapsulated into components. Not only does the **channel** or **bridge** component provide a connection between the standard components, but it also provides information on the addressing and data flow. With this information, it is possible to construct things such as a memory map for the system.

A *channel* connects component master, slave, and system interfaces on the same bus. All masters connected to a channel see all slaves at the same physical address and only one transaction can be active in a channel at a time. This does not preclude bus protocols that utilize pipelining.

A *bridge* is an interface between one bus and another (often a peripheral bus to the main system bus). Such a component has at least one master interface (onto the peripheral bus) and one slave interface (onto the main system bus). Crossbar bus infrastructure (e.g., an ARM Multilayer AMBA) is also treated as a bus bridge—

such examples might have multiple master and multiple slave interfaces. A bridge can support multiple simultaneous transactions and the slaves existing in the master interface address spaces may appear at different address to any masters connected (by a channel) to each of the bus bridge's slave ports.

7.4.1 Channel

The channel is a general name which denotes the collection of connections between multiple internal bus interfaces. The memory map between these connections is restricted so that, for example, a generator can be called to automatically compute all the address maps for the complete design. A channel can represent a simple wiring interconnect or a more complex structure such as a bus.

A channel also encapsulates the connection between master and slave components. A channel is the construct, which represents the bus infrastructure and allows transactions initiated by a master interface to be completed by a slave interface.

The following rules apply for using channels.

a) A channel can only have one address space (i.e., transmission/transformation matrix). In other words, a slave connected to a channel has the same address as seen from all masters connected to this channel. This guarantees the slave addresses (as seen by each master) are consistent for the system. As a consequence, all slave interfaces connected to a channel see the same address (if they do not, they are connected to different channels); and if more than one master/slave interface pair is active or selected simultaneously, there is more than one channel present.

b) A channel can only relate mirrored interfaces because some buses can have asymmetric interfaces (e.g., AHB). To cover all type of buses, the channel interfaces are always mirrored interfaces. As a consequence, a channel can only connect to a direct interface (it can not connect directly to another channel). However, not all mirrored interfaces of a channel need to be connected.

c) A channel cannot be hierarchical.

d) A channel supports memory mapping and re-mapping (see 7.8 and 7.9).

Simple wire connections (e.g., a clock port connecting to all components of the system) may be modeled as an IP-XACT **channel** or as IP-XACT **port** object.

The following is a sample of the XML code describing the **channel** and its mirrored interfaces for a simple AHB-like bus component.

```
<spirit:component>
   …
  <spirit:busInterfaces>
    <spirit:busInterface spirit:id="AHB_MS">
      <spirit:name>AHB_mirror_slave</spirit:name>
      <spirit:busType spirit:library="AMBA" spirit:name="simpleAHB"
   spirit:vendor="spiritconsortium.org" />
      <spirit:mirroredSlave/>
      <spirit:connection>required</spirit:connection>
    <spirit:busInterface spirit:id="AHB_MM">
      <spirit:name>AHB_mirror_master</spirit:name>
      <spirit:busType spirit:library="AMBA" spirit:name="simpleAHB"
   spirit:vendor="spiritconsortium.org" />
      <spirit:mirroredMaster/>
    </spirit:busInterface>
  </spirit:busInterfaces>

  <spirit:channels>
     <spirit:channel>
```

```
        <spirit:name>channelAHB1</spirit:name>
        <spirit:busInterfaceRef>AHB_mirror_slave</spirit:busInterfaceRef>
        <spirit:busInterfaceRef>AHB_mirror_master</spirit:busInterfaceRef>
      </spirit:channel>
    </spirit:channels>

  </spirit:component>
```

7.4.2 Bridge

Some buses can be modeled using component bridges. The bridge is a mechanism to model the internal relationship between slave interfaces and master interfaces inside a component. The slave interface in a bridge is the interface where a transaction arrives and the master interface is the interface where the transaction exits. There two different types of bridges defined in IP-XACT, a transparent bridge (**opaque**=”false”) and an opaque bridge (**opaque**=”true”).

The following rules apply for using bridges.

a) A bridge can have multiple address spaces. Specifically, a bridge shall have one or more master interfaces and each master interface may have a local address space associated with that interface.

b) A bridge can only have direct interfaces. As a consequence, a bridge can directly connect to another component (master interface to slave interface connection) under the conditions defined in section 4.8.3.2. Or it can connect to a channel (e.g., master interface to mirrored-master interface).

c) A bridge can be hierarchical.

d) A bridge supports memory mapping and re-mapping (see 7.8 and 7.9).

In a bridge, multiple transactions can occur simultaneously, e.g., if two slave interfaces receive a transaction addressing two distinct master interfaces who want to access the bus at the same time, both can be granted as long as a ‘bridge path’ has been defined in IP-XACT.

7.4.2.1 Transparent bridge

Needs to be written

7.4.2.2 Opaque bridge

Needs to be written

7.4.3 Combining channels and bridges

It is possible to combine channels and bridges together each in separate components to form a new hierarchical component for the purpose of modeling more complex interconnects. A multi-layer bus is a more complex interconnect which may have multiple transactions active and support multiple memory maps. As such, it cannot be modeled as a channel and if the interfaces are asymmetric (they do not allow direct connections), then the bus also cannot be modeled as a bridge.

The solution is to use a combination of channel and bridge components. The bridge component in the center forms the main cross-bar for the communications between components. It decides which interfaces may bridge to other interfaces. The smaller channels then come in to convert the direct interface of the bridge (which could not connect to the master’s or slave’s because of the asymmetric bus) into a mirrored interface that can now connect with a direct-mirrored connection to the master or slave. An example of this is shown in Figure 9.

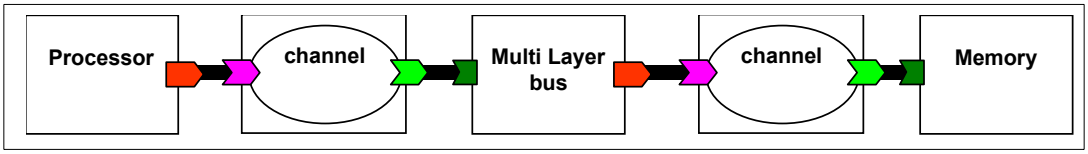


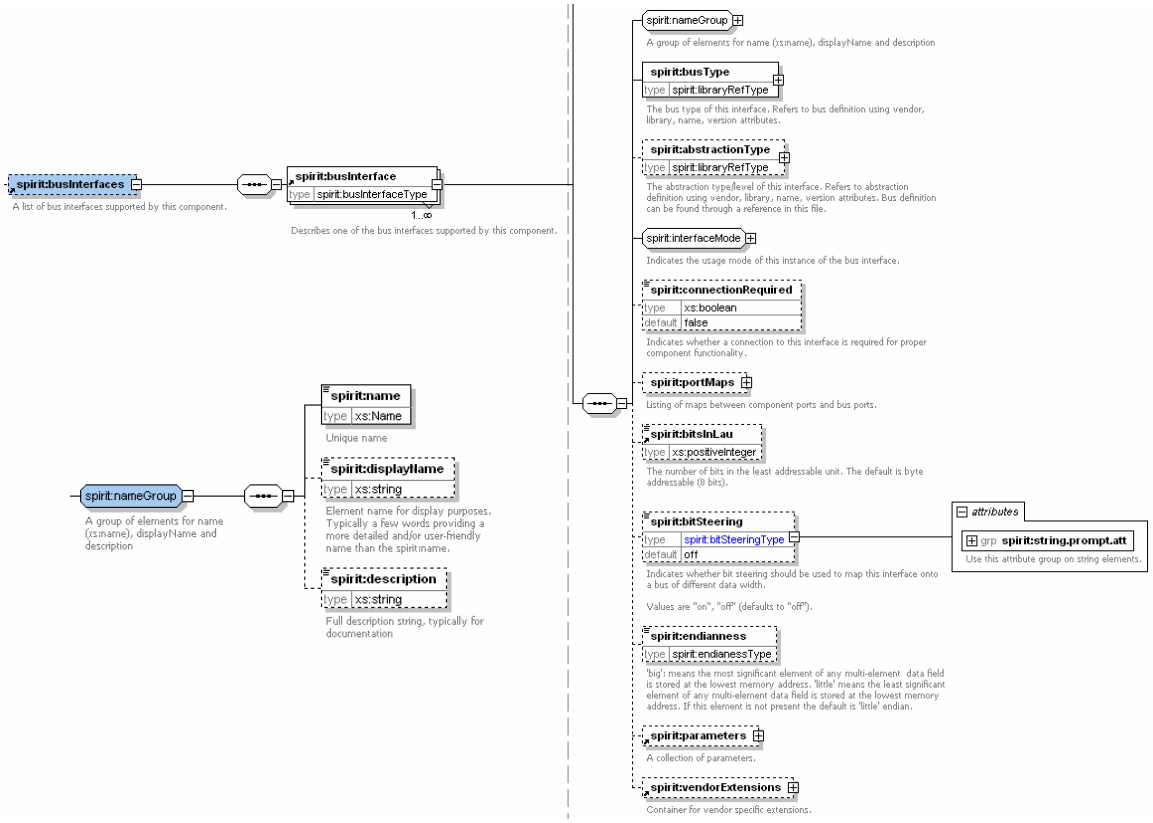
Figure 9—Asymmetric multi-layer bus connection using channels

7.5 Bus interfaces

7.5.1 busInterface

7.5.1.1 Schema

The following schema details the information contained in the **busInterfaces** element, which may appear as an element inside the top-level **component** element.



7.5.1.2 Description

Bus interfaces enable individual ports that appear on the component to be grouped together into a meaningful, known protocol. When the protocol is known, a lot of additional information can be written down about the characteristics of that interface.

The **busInterfaces** element contains an unbounded list of **busInterface** elements; therefore, a **component** may have multiple bus interfaces of the same or different types. Each **busInterface** element defines properties of this specific interface in a component. It contains the following elements and attributes.

a) ***nameGroup*** group includes the following. See X.Y.Z.

   1) **name** (mandatory) identifies a name for the bus interface.

   2) **displayName** (optional) allows a short descriptive text to be associated with the bus interface.

   3) **description** (optional) allows a textual description of the bus interface.

b) **busType** (mandatory) specifies the bus definition that this bus interface is referenced. A bus definition (see 6.2) describes the high-level attributes of a bus description. The **busType** element is of type ***libraryRefType*** (see X.Y.Z); it contains four attributes to specify a unique VLNV.

   1) The **vendor** attribute (mandatory) identifies the owner of the referenced description.

   2) The **library** attribute (mandatory) identifies a library of the referenced description.

   3) The **name** attribute (mandatory) identifies a name of the referenced description.

   4) The **version** attribute (mandatory) identifies a version of the referenced description.

c) **abstractionType** (mandatory) specifies the abstraction definition where this bus interface is referenced. An abstraction definition describes the low-level attributes of a bus description (see 6.3). The **abstractionType** element is of type ***libraryRefType*** (see X.Y.Z); it contains four attributes to specify a unique VLNV.

   1) The **vendor** attribute (mandatory) identifies the owner of the referenced description.

   2) The **library** attribute (mandatory) identifies a library of the referenced description.

   3) The **name** attribute (mandatory) identifies a name of the referenced description.

   4) The **version** attribute (mandatory) identifies a version of the referenced description.

d) ***interfaceMode*** group describes further information on the *mode* for this interface. There are seven possible modes for an interface: master, slave, mirroredMaster, mirroredSlave, system, mirroredSystem and monitor. See X.Y.Z for details on the ***interfaceMode*** group.

e) **connectionRequired** (optional), if ***True***, specifies when this component is integrated; this interface must be connected to another interface for the integration to be valid. If ***False*** (the default) this interface may be left unconnected. The **connectionRequired** element is of type ***Boolean***.

f) **portMaps** (optional) describes the mapping between the abstraction definition's logical ports and the component's physical ports. See 7.5.2.7.

g) **bitsInLau** (optional) describes the number of data bits that are addressable by the least significant address bit in the bus interface. It is only appropriate to specify this element for interfaces that are addressable. The **bitsInLau** element is of type ***positiveInteger***. The default value is **8**.

h) **bitSteering** (optional) designates if this interface has the ability to dynamically align data on different byte channels on a data bus. This element shall only be specified for interfaces that are addressable. The **bitSteering** element is a choice of two values, ***on*** indicating this interface uses data steering logic and ***off*** that this interface does not use data steering logic. The **bitSteering** element is configurable, using attributes from ***string.prompt.att***, see X.Y.Z on configuration.

i) **endianness** (optional) indicates the endianness of the bus interface. The two choices are ***big*** for big-endian and ***little*** for little-endian. For further information on endianness, see 7.5.1.2.1. This element shall only be specified for interfaces that are addressable.

j) **parameters** (optional) specifies any parameter data value(s) for this bus interface.

k) **vendorExtensions** (optional) holds any vendor-specific data from other name spaces which is applicable to this bus interface.

7.5.1.2.1 Endianness

Endianness is defined under the **busInterface** element of the component. There are (only) two legal values (***big*** and ***little***) to specify the **endianness**.

— *Big endian* (**big**) means the most significant byte of any multi-byte data field is stored at the lowest memory address, which is also the address of the larger field.

— *Little endian* (**little**) means the least significant byte of any multi-byte data field is stored at the lowest memory address, which is also the address of the larger field.

7.5.1.2.2 Big-endianness

There are at least two ways for big-endianness to manifest itself, byte-invariant and word-invariant (also known as *middle-endian*); the difference being if data is stored as *word-invariant*, the data is stored differently for transfers larger than a byte, e.g.,

a) Byte invariant: A word access to address `0x0` is on `D[31:0]`. The MSB is `D[7:0]`, the LSB is `D[31:24]`.

b) Word invariant: A word access to address `0x0` is on `D[31:0]`. The MSB is `D[31:24]`, the LSB byte is `D[7:0]`.

c) In IP-XACT, the interpretation of big-endian is the byte-invariant style.

7.5.1.3 Example

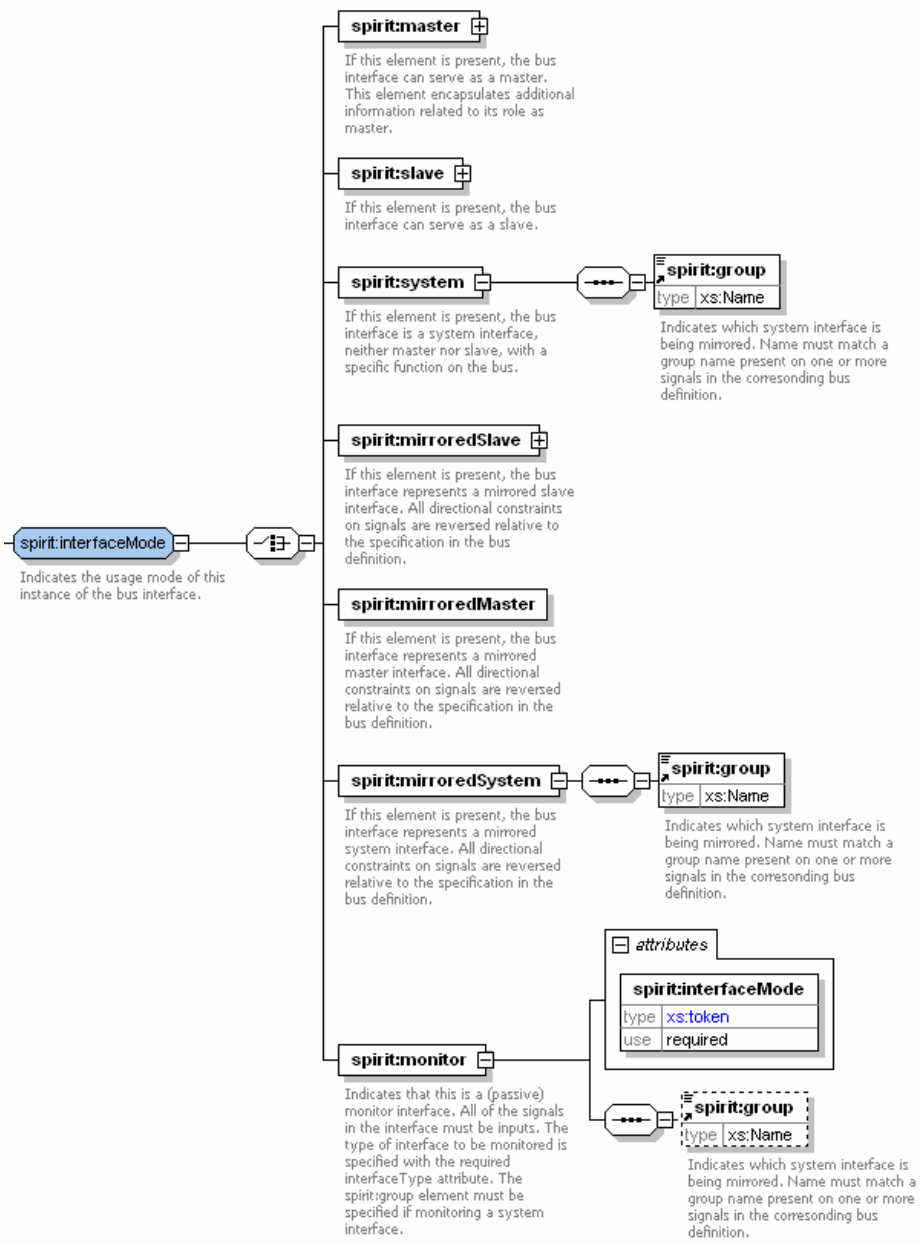
The example below shows a simple bus interface for a clock signal. The interface reference a bus definition and an abstraction definition.

```
<spirit:busInterface>
   <spirit:name>APBClk</spirit:name>
   <spirit:busType spirit:vendor="spiritconsortium.org"
   spirit:library="busdef.clock" spirit:name="clock" spirit:version="1.0"/>
   <spirit:abstractionType spirit:vendor="spiritconsortium.org"
   spirit:library="busdef.clock" spirit:name="clock_rtl"
   spirit:version="1.0"/>
   <spirit:slave/>
   <spirit:portMaps>
      <spirit:portMap>
         <spirit:logicalPort>
            <spirit:name>CLK</spirit:name>
         </spirit:logicalPort>
         <spirit:physicalPort>
            <spirit:name>clk</spirit:name>
         </spirit:physicalPort>
      </spirit:portMap>
   </spirit:portMaps>
</spirit:busInterface>
```

7.5.2 Interface modes

The following schema details the information contained in the ***interfaceMode*** group, which appears as a group inside the **busInterface** element.

7.5.2.1 Schema



7.5.2.2 Description

The **busInterface**'s mode designates the purpose of the **busInterface** on this component. There are seven possible modes: three pairs of standard functional interfaces and their mirrored counterparts, and a monitor interface for VIP.

The ***interfaceMode*** group shall contain one of the following seven elements.

a) A **master** interface mode (sometimes also known as an *initiator*) is one that initiates transactions. See 7.5.2.4.

b) A **slave** interface mode (sometimes also known as a *target*) is one that responds to transactions.

c) A **system** interface mode is used for some classes of interface that are standard on different bus types, but do fit into the master or slave category.

The **group** (mandatory) attribute for the **system** element defines the name of the group to which this system interface belongs. The type of the group attribute is ***Name***. The specified value of **group** needs to be a group defined in the referenced abstraction definition. A connection between a **system** and **mirroredSystem** interfaces shall have matching group names.

d) A **mirroredSlave** interface mode is the mirrored version of a slave interface and can provide addition address offsets to the connected slave interface. See 7.5.2.6

e) A **mirroredMaster** interface mode is the mirrored version of a master interface.

f) A **mirroredSystem** interface mode is the mirrored version of a system interface.

The **group** (mandatory) attribute for the **mirroredSystem** element defines the name of the group to which this **mirroredSystem** interface belongs. The type of the group attribute is ***Name***. The specified value of **group** needs to be a group defined in the referenced abstraction definition. A connection between a **system** and **mirroredSystem** interfaces shall have matching group names.

g) A **monitor** interface mode is a special interface that can be used for verification. This monitor interface mode is used to gather data from other interfaces. A monitor may only connect to interfaces that match its set **interfaceMode**. See 7.3.3.

1) The **interfaceMode** (mandatory) attribute defines the interface mode for which this monitor interface can be connected.: ***master***, ***slave***, ***system***, ***mirroredMaster***, ***mirroredSlave***, *or* ***mirroredSystem***.

2) The **group** (optional) element is required if the **interfaceMode** attribute is set to ***system*** or ***mirroredSystem***. This element defines the name of the system group for this monitor interface. The type of the **group** element is ***Name***. The specified value of **group** shall be a group defined in the referenced abstraction definition.

7.5.2.3 Example

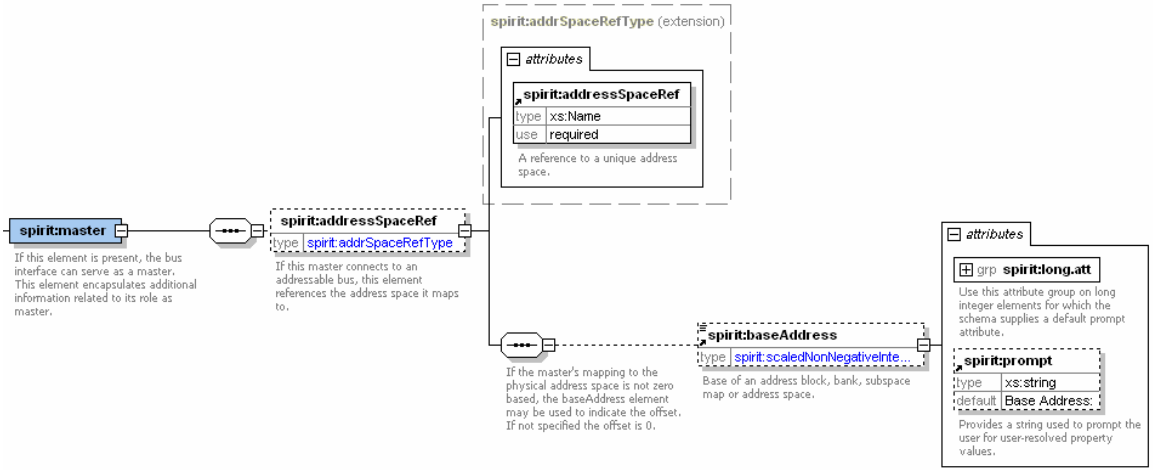
The example below shows a portion of a bus interface for an AHB bus interface. The interface mode is defined as monitor for a slave.

```
<spirit:busInterface>
    <spirit:name>ambaAHBSlaveMonitor</spirit:name>
    <spirit:busType spirit:vendor="amba.com" spirit:library="AMBA2"
    spirit:name="AHB" spirit:version="r2p0_5"/>
    <spirit:abstractionType spirit:vendor="amba.com" spirit:library="AMBA2"
    spirit:name="AHB_rtl" spirit:version="r2p0_5"/>
    <spirit:monitor spirit:interfaceMode="slave"/>
        <spirit:portMaps>
            <spirit:portMap>
                <spirit:logicalPort>
                    <spirit:name>HRESP</spirit:name>
                </spirit:logicalPort>
                <spirit:physicalPort>
                    <spirit:name>hresp</spirit:name>
                </spirit:physicalPort>
            </spirit:portMap>
...
</spirit:busInterface>
```

7.5.2.4 Master interface

The following schema details the information contained in the **master** element, which appears as an element inside the ***interfaceMode*** group inside **busInterface** element.

7.5.2.4.1 Schema



7.5.2.4.2 Description

A **master** interface (sometimes also known as an initiator) is one that initiates transactions. The **master** element contains the following elements and attributes.

a) **addressSpaceRef** (optional) element contains attributes and subelements to describe information about the range of addresses with which this master interface can generate transactions. If the interface is a bus definition that is addressable, an address space reference shall be included.

   1) **addressSpaceRef** (mandatory) attribute references a name of an address space defined in the same component. The address space shall define the range and width for transaction on this interface. See 7.7.

   2) **baseAddress** (optional) specifies the starting address of the address space. The address space numbering normally starts at `0`. Some address spaces may use *offset addressing* (starting at a number other than `0`) so the base address element can be used to designate this information. The type of this element is set to ***scaledNonNegativeInteger***, see C.10. The **baseAddress** element is configurable, with attributes from ***long.att***, see X.Y.Z on configuration. The **prompt** (optional) attribute allows the setting of a string for the configuration and has a default value of **"Base Address:"**.

7.5.2.4.3 Example

The example below shows a portion of a bus interface for an AHB master bus interface. The interface contains a reference to an address space called `main`, that has its base address starting at `0`.

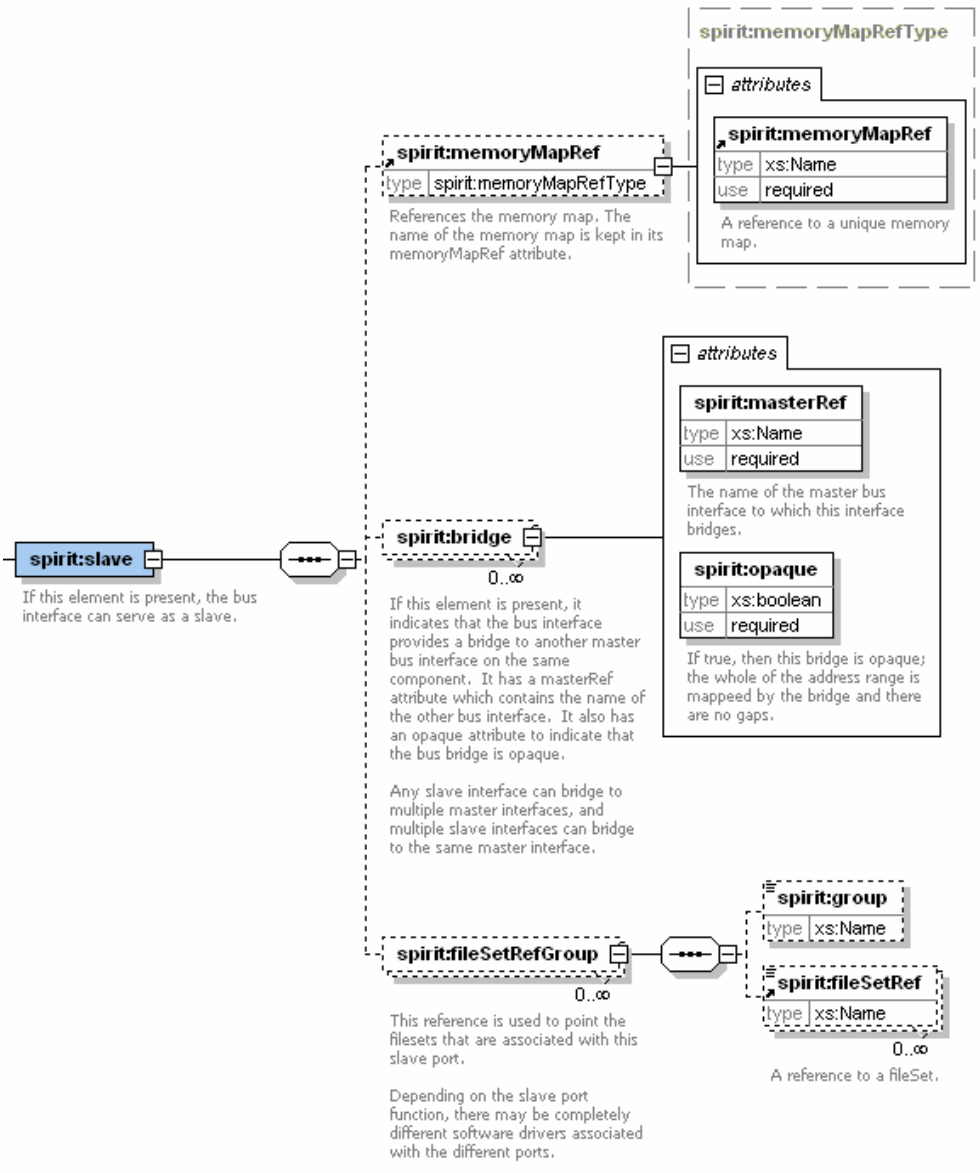
```
<spirit:busInterface>
   <spirit:name>AHBmaster</spirit:name>
   <spirit:busType spirit:vendor="amba.com" spirit:library="AMBA2"
   spirit:name="AHB" spirit:version="r2p0_5"/>
   <spirit:abstractionType spirit:vendor="amba.com" spirit:library="AMBA2"
   spirit:name="AHB_rtl" spirit:version="r2p0_5"/>
```

```
        <spirit:master>
        <spirit:addressSpaceRef spirit:addressSpaceRef="main"/>
        </spirit:master>
        <spirit:connectionRequired>true</spirit:connectionRequired>
        <spirit:portMaps>
            <spirit:portMap>
            <spirit:logicalPort>
                <spirit:name>HRDATA</spirit:name>
            </spirit:logicalPort>
            <spirit:physicalPort>
                <spirit:name>hrdata</spirit:name>
            </spirit:physicalPort>
        </spirit:portMap>
    ...
    </spirit:busInterface>
```

7.5.2.5 Slave interface

The following schema details the information contained in the **slave** element, which appears as an element inside the ***interfaceMode*** group inside **busInterface** element.

7.5.2.5.1 Schema



7.5.2.5.2 Description

A **slave** interface (sometimes also known as a target) is one that responds to transactions. The memory map reference points to information about the range of registers, memory, or other address blocks accessible through this slave interface. This slave interface can also be used in a bridge application to “bridge” a transaction from a slave interface to a master interface.

a) **memoryMapRef** (optional) element contains an attribute that references an memory map. If the interface is a bus definition that is addressable, a **memoryMapRef** element shall be included, unless the slave interface is part of a **bridge** with **opaque**=***False***.

The **memoryMapRef** (mandatory) attribute references a name of a memory map defined in the same component. The memory map contains information about the range of registers, memory, or other address blocks. See 7.8.

b) **bridge** (optional) element is an unbounded list of references to master interfaces. If the interface is of a bus definition that is addressable, a bridge element may be included.

1) The **masterRef** (mandatory) attribute shall reference a master interface in the containing component. Under some conditions, transactions from the slave interface may be bridged to the referenced master interface, as defined by **opaque** (see also 7.4.2).

2) The **opaque** (mandatory) attribute defines the type of bridging. The **opaque** attribute is of type ***Boolean***. ***True*** means the addressing entering into the slave interface shall have the subspace maps **baseAddress** subtracted and, if non-negative, the result shall exit on the subspace maps' referenced master interface's referenced address space. ***False*** means all addressing entering the slave interface shall exit the above referenced master interface without any modifications, this type of bridge is sometimes called *transparent*.

c) **fileSetRefGroup** (optional) element is an unbounded list of the references to file sets contained in this component. These file set references are associated with this slave interface. This element may seem out of place, but it allows each slave port to reference a unique **fileSet** element (see 7.13). This element can further be used to reference a software driver, which can be made different for each slave port.

1) **group** (optional) element allows the definition of a group name for the **fileSetRefGroup**. The **group** element is of type ***Name***.

2) **fileSetRef** (optional) element is an unbounded list of references to a **fileSet** element contained in this component. The **fileSetRef** element is of type ***Name***. See 7.13.

7.5.2.5.3 Example

The example below shows a portion of an opaque bridge from and AHB slave bus interface to an APB master bus interface.

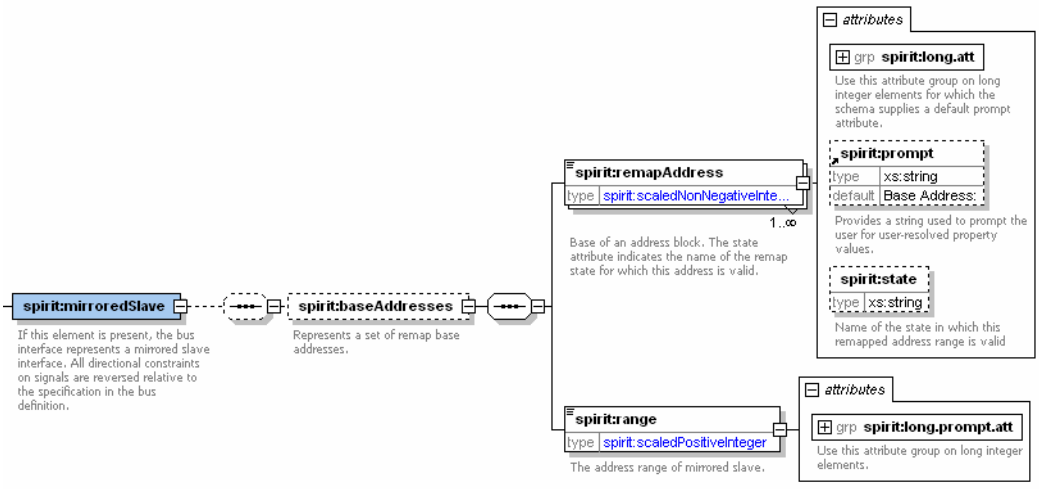
```
<spirit:busInterface>
    <spirit:name>ambaAPB</spirit:name>
    <spirit:busType spirit:vendor="amba.com" spirit:library="AMBA2"
    spirit:name="APB" spirit:version="r2p0_3"/>
    <spirit:abstractionType spirit:vendor="amba.com" spirit:library="AMBA2"
    spirit:name="APB_rtl" spirit:version="r2p0_3"/>
    <spirit:master>
        <spirit:addressSpaceRef spirit:addressSpaceRef="apb"/>
    </spirit:master>
...
<spirit:busInterface>
    <spirit:name>ambaAHB</spirit:name>
    <spirit:busType spirit:vendor="amba.com" spirit:library="AMBA2"
    spirit:name="AHB" spirit:version="r2p0_5"/>
    <spirit:abstractionType spirit:vendor="amba.com" spirit:library="AMBA2"
    spirit:name="AHB_rtl" spirit:version="r2p0_5"/>
    <spirit:slave>
        <spirit:memoryMapRef spirit:memoryMapRef="ambaAHB"/>
        <spirit:bridge spirit:masterRef="ambaAPB" spirit:opaque="true"/>
    </spirit:slave>
...
<spirit:addressSpaces>
    <spirit:addressSpace>
        <spirit:name>apb</spirit:name>
```

```
        <spirit:range spirit:choiceRef="addressWidthChoice"
    spirit:format="choice" spirit:id="masterRange" spirit:prompt="Master Port
    Size :" spirit:resolve="user">1M</spirit:range>
        <spirit:width spirit:format="long">32</spirit:width>
    </spirit:addressSpace>
</spirit:addressSpaces>

<spirit:memoryMaps>
    <spirit:memoryMap>
        <spirit:name>ambaAHB</spirit:name>
        <spirit:subspaceMap spirit:masterRef="ambaAPB">
            <spirit:name>bridgemap</spirit:name>
            <spirit:baseAddress>0x10000000</spirit:baseAddress>
        </spirit:subspaceMap>
    </spirit:memoryMap>
</spirit:memoryMaps>
```

7.5.2.6 Mirrored slave interface

The following schema details the information contained in the **mirroredSlave** element, which appears as an element inside the ***interfaceMode*** group inside **busInterface** element.

7.5.2.6.1 Schema



7.5.2.6.2 Description

A **mirroredSlave** interface is used to connect to a **slave** interface. The **mirroredSlave** interface may contain additional address information in the **baseAddresses** (optional) element.

a) **remapAddress** (mandatory) element is an unbounded list that specifies the address offset to apply to the connected slave interface. The type of this element is set to ***scaledNonNegativeInteger***, see C.10. The **remapAddress** element is configurable with attributes from ***long.att***, see X.Y.Z on configuration. The **prompt** (optional) attribute allows the setting of a string for the configuration and has a default value of **"Base Address:"**. The **state** (optional) attribute references a defined state in the component and identifies the remap state name for which the **remapAddress** and **range** apply. See 7.9.2.

b) **range** (mandatory) specifies the address range to apply to the connected **slave** interface. The **range** is expressed as the number of addressable units based on the size of an addressable unit is defined inside the containing **busInterface**/**bitsInLau** element. See 7.5.1. The type of this element is set to ***scaledPositiveInteger***. The **range** element is configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

7.5.2.6.3 Example

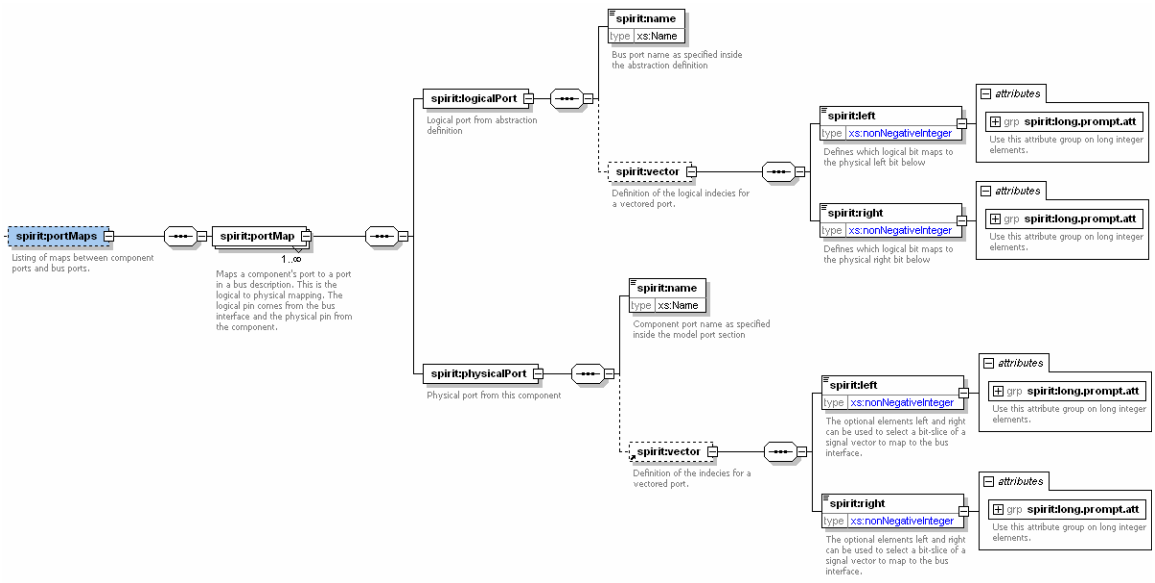
This example shows a portion of a bus interface for an AHB mirroredSlave bus interface. The interface contains two remap addresses. The first does not have a state attribute and is always active unless a named state is active, in this case, the base address of the connected slave is offset by `0x00000000`. The second remap address is active when state equal `reampped` is selected, in this case the base address of the slave is offset by `0x10000000`.

```
<spirit:busInterface>
    <spirit:name>MirroredSlave0</spirit:name>
    <spirit:busType spirit:vendor="amba.com" spirit:library="AMBA2"
    spirit:name="AHB" spirit:version="r2p0_5"/>
    <spirit:abstractionType spirit:vendor="amba.com" spirit:library="AMBA2"
    spirit:name="AHB_rtl" spirit:version="r2p0_5"/>
    <spirit:mirroredSlave>
        <spirit:baseAddresses>
            <spirit:remapAddress spirit:resolve="user"
    spirit:id="start_addr_slv0_mirror" spirit:choiceRef="BaseAddressChoices"
    spirit:format="choice" spirit:prompt="Slave 0 Starting
    Address:">0x00000000</spirit:remapAddress>
            <spirit:remapAddress spirit:resolve="user"
    spirit:id="restart_addr_slv0_mirror"
    spirit:choiceRef="BaseAddressChoices" spirit:format="choice"
    spirit:prompt="Remap Slave 0 Starting Address:"
    spirit:state="remapped">0x10000000</spirit:remapAddress>
            <spirit:range spirit:resolve="user" spirit:id="range_slv0_mirror"
    spirit:prompt="Slave 0 Range:">0x00010000</spirit:range>
        </spirit:baseAddresses>
    </spirit:mirroredSlave>
...
</spirit:busInterface>
```

7.5.2.7 Port mapping

The following schema details the information contained in the **portMaps** element, which appears as an element inside **busInterface** element.

7.5.2.7.1 Schema



7.5.2.7.2 Description

The **portMaps** (optional) element contains an unbounded list of **portMap** elements. Each **portMap** element describes the mapping between the logical ports, defined in the referenced abstraction definition, to the physical ports, defined in the containing component description.

a) **logicalPort** (mandatory) contains the information on the logical port from the abstraction definition.

   1) **name** (mandatory) specifies the logical port name. The name shall be a name of a logical port in the referenced abstraction definition that is defined as legal for this interface mode. The **name** element is of type ***Name***.

   2) **vector** (optional) is used for a vectored logical port to specify the indices of the logical port mapping. The **vector** element contains two subelements: **left** and **right**. The values of **left** and **right** shall be less than the **width** if specified for the logical port from the abstraction definition. The **left** and **right** elements are both of type ***nonNegativeInteger***. The **left** and **right** elements are configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

b) **physicalPort** (mandatory) contains information on the physical port contained in the component.

   1) **name** (mandatory) specifies the physical port name. The name shall be a name of a port in the containing component. The **name** element is of type ***Name***.

   2) **vector** (optional) is used for a vectored physical port to specify the indices of the physical port mapping. The **vector** element contains two subelements: **left** and **right**. The values of **left** and **right** shall be within the **left** and **right** values specified for the physical port. The **left** and **right** elements are both of type ***nonNegativeInteger***. The **left** and **right** elements are configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

The same physical port may be mapped to a number of different logical ports on the same or different bus interfaces, and the same logical port may be mapped to a number of different physical ports. For port mapping rules, see 7.3.4.1.

7.5.2.7.3 Example

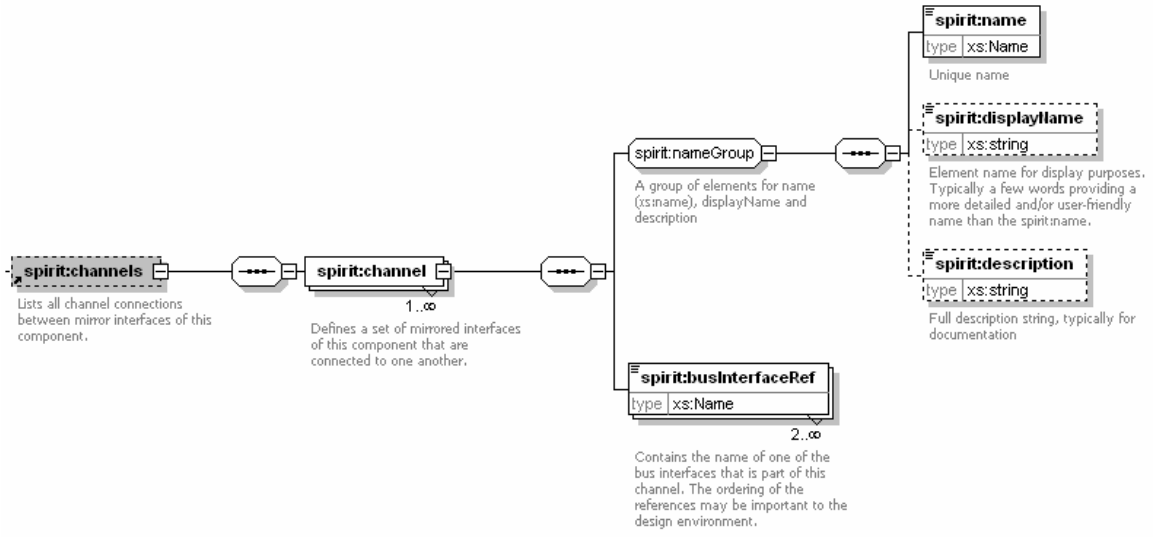
The example below shows a portion of a bus interface for an APB bus interface. A mapping from the logical port `PADDR` to the lower 12 bits of the physical port `paddr`. A mapping from the logical port `PWRITE` to the physical port `pwrite`.

```
<spirit:portMap>
   <spirit:logicalPort>
      <spirit:name>PADDR</spirit:name>
   </spirit:logicalPort>
   <spirit:physicalPort>
      <spirit:name>paddr</spirit:name>
      <spirit:vector>
         <spirit:left>11</spirit:left>
         <spirit:right>11</spirit:right>
      </spirit:vector>
   </spirit:physicalPort>
</spirit:portMap>
<spirit:portMap>
   <spirit:logicalPort>
      <spirit:name>PWRITE</spirit:name>
   </spirit:logicalPort>
   <spirit:physicalPort>
      <spirit:name>pwrite</spirit:name>
   </spirit:physicalPort>
</spirit:portMap>
```

7.6 Component channels

7.6.1 Schema

The following schema details the information contained in the **channels** element, which may appear as an element inside the top-level **component** element.



7.6.2 Description

The **channels** element contains an unbounded list of **channel** elements. Each **channel** element contains a list of all the mirrored bus interfaces in the containing component that belong to the same channel.

a) ***nameGroup*** group includes the following. See X.Y.Z.

   1) **name** (mandatory) identifies the channel.

   2) **displayName** (optional) allows a short descriptive text to be associated with the channel.

   3) **description** (optional) allows a textual description of the channel.

b) **busInterfaceRef** (mandatory) is an unbound list of references (a minimum of two) to mirrored bus interfaces in the containing component. Each mirrored bus interface in a component may be referenced in any channel at most once. The order of this list may be used by the design environment in some way and shall be maintained. The **busInterfaceRef** element is of type ***Name***.

The referenced **busInterface**s need to be compatible, which implies the underlying **busDefinition**s (referenced by VLNV) need to be compatible as well. The maximum number of mirrored-master interfaces that can be connected to a channel is determined by the smallest value of **maxMasters** in the **busDefinition**s of the referenced **busInterface**s. The maximum number of mirrored-slave interfaces is likewise determined by the corresponding **maxSlaves** values.

See also: SCR 3.1, SCR 3.2, SCR 3.3, SCR 3.4, and SCR 3.5.

7.6.3 Example

The following example shows a channel with two connected **busInterface**s.

```
<spirit:busInterfaces>
  <spirit:busInterface>
    <spirit:name>InterfaceA</spirit:name>
    <spirit:busType>...</spirit:busType>
    <spirit:master>...</spirit:master>
  </spirit:busInterface>
  <spirit:busInterface>
    <spirit:name>InterfaceB</spirit:name>
    <spirit:busType>...</spirit:busType>
    <spirit:slave>...</spirit:slave>
  </spirit:busInterface>
</spirit:busInterfaces>

<spirit:channels>
  <spirit:channel>
    <spirit:name>masterChannel</spirit:name>
    <spirit:displayName>Channel for Master communication</spirit:displayName>
    <spirit:description>This channel includes all transaction calls used by
   the master component of the system</spirit:description>
    <spirit:busInterfaceRef>InterfaceA</spirit:busInterfaceRef>
    <spirit:busInterfaceRef>InterfaceB</spirit:busInterfaceRef>
  </spirit:channel>
</spirit:channels>
```

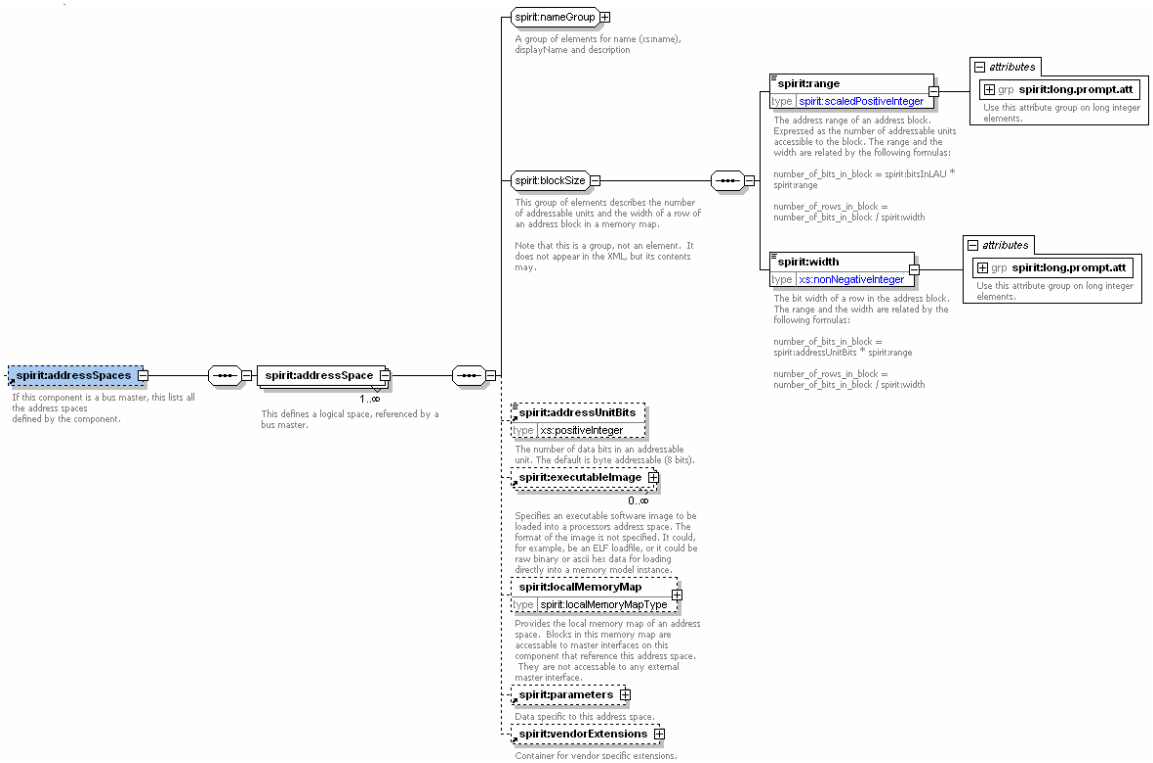
7.7 Address space

An address space is defined as a logical addressable space of memory. Each master interface can be assigned a logical address space. Address spaces are effectively the programmer's view looking out from a master port. Some components may have address spaces associated with more than one master interface (for instance, a processor that has a system bus and a fast memory bus. Other components (for instance, Harvard architecture processors) may have multiple address spaces - one for instruction and the other for data.

7.7.1 addressSpaces

7.7.1.1 Schema

The following schema details the information contained in the **addressSpaces** element, which may appear as an element inside the top-level **component** element.



7.7.1.2 Description

The **addressSpaces** element contains an unbouded list of **addressSpace** elements. Each **addressSpace** element defines a logical address space seen by a master bus interface. It contains the following elements.

a) ***nameGroup*** group includes the following. See X.Y.Z.
   1) **name** (mandatory) identifies the address space.
   2) **displayName** (optional) allows a short descriptive text to be associated with the address space.
   3) **description** (optional) allows a textual description of the address space.

b) ***blockSize*** group includes the following.
   4) **range** (mandatory) gives the address range of an address space. This is expressed as the number of addressable units of the address space. The size of an addressable unit is defined inside

the **addressUnitBits** element. The type of the **range** element is set to scaledPositiveInteger. The **range** element is configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

5) **width** (mandatory) is the bit width of a row in the address space. The type of this element is set to nonNegativeInteger. The **width** element is configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

c) The optional **addressUnitBits** elements defines the number of data bits in each address increment of the address space.

d) **executableImage** (optional) describes the details of an executable image that can be loaded and executed in this address space on the processor to which this master bus interface belongs.

e) **localMemoryMap** (optional) describes a local memory map that is seen exclusively by this master bus interface viewing this address space. See 7.7.6.

f) **parameters** (optional) specifies any parameter data value(s) for this address space.

g) **vendorExtensions** (optional) holds any vendor-specific data from other name spaces which is applicable to this address space.

7.7.1.3 Example

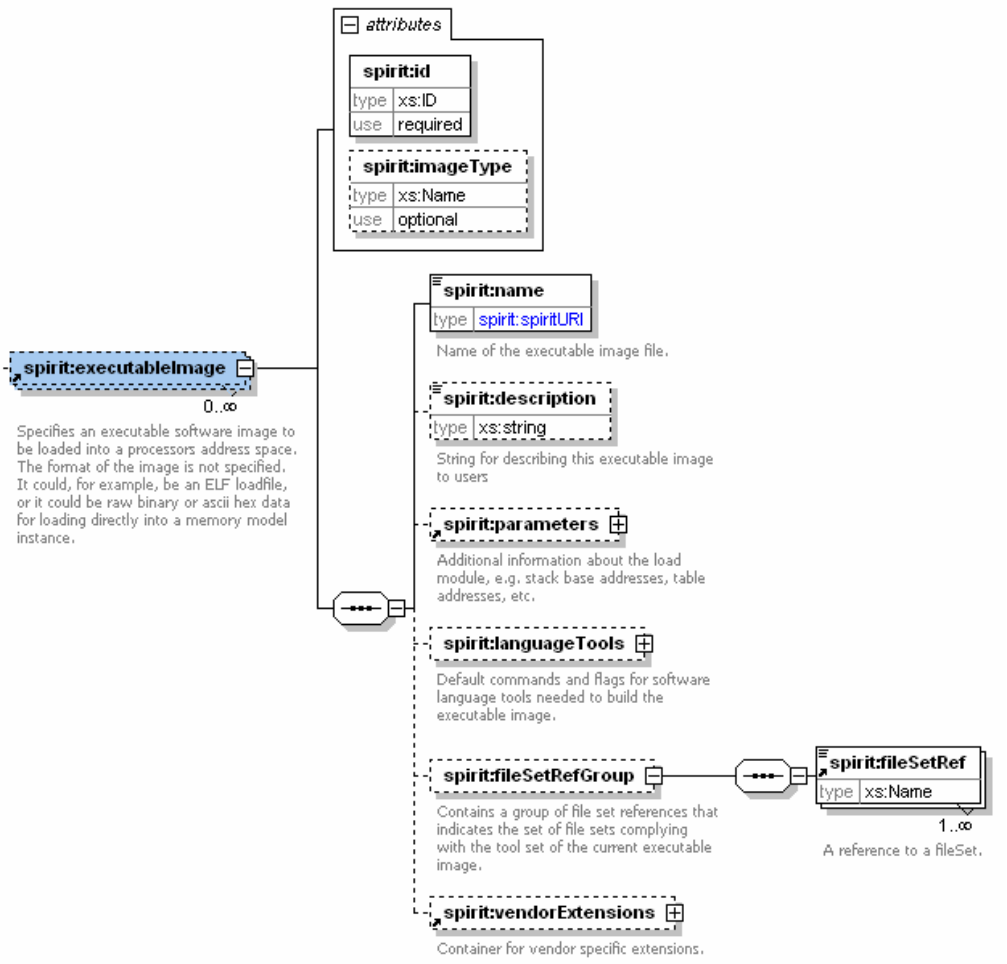
The following example shows the definition of an address space with a range (length) of 4 giga-bytes and a width of 32 bits.

```
<spirit:addressSpaces>
   <spirit:addressSpace>
      <spirit:name>main</spirit:name>
      <spirit:range>4G</spirit:range>
      <spirit:width>32</spirit:width>
      <spirit:addressUnitBits>8</spirit:addressUnitBits>
   </spirit:addressSpace>
</spirit:addressSpaces>
```

7.7.2 executableImage

7.7.2.1 Schema

The following schema details the information contained in the **executableImage** element, which may appear inside an **addressSpace** element.



7.7.2.2 Description

The **executableImage** element contains a list of further elements.

a) **id** (mandatory) attribute uniquely identifies the **executableImage** for reference else where in this description, reference location unknown.

b) **imageType** (optional) attribute can describe the binary executable format (e.g., ELF, raw binary, etc.). The list of possible values is user defined.

c) **name** (reqired) identifies the location of the executable object. The type is spiritURI.

d) **description** (optional) allows a textual description of the address space.

e) **parameters** (optional) specifies any parameter data value(s) for this executable object.

f) **languageTools** (optional) contains further elements to describe the information need to build the execuable image. See 7.7.3.

g) **fileSetRefGroup** (optional) element contains a list of **fileSetRef** subelements, each one containing the name of a file set associated with this **executableImage**.

h) **vendorExtensions** (optional) holds any vendor-specific data from other name spaces which is applicable to this address space.

7.7.2.3 Example

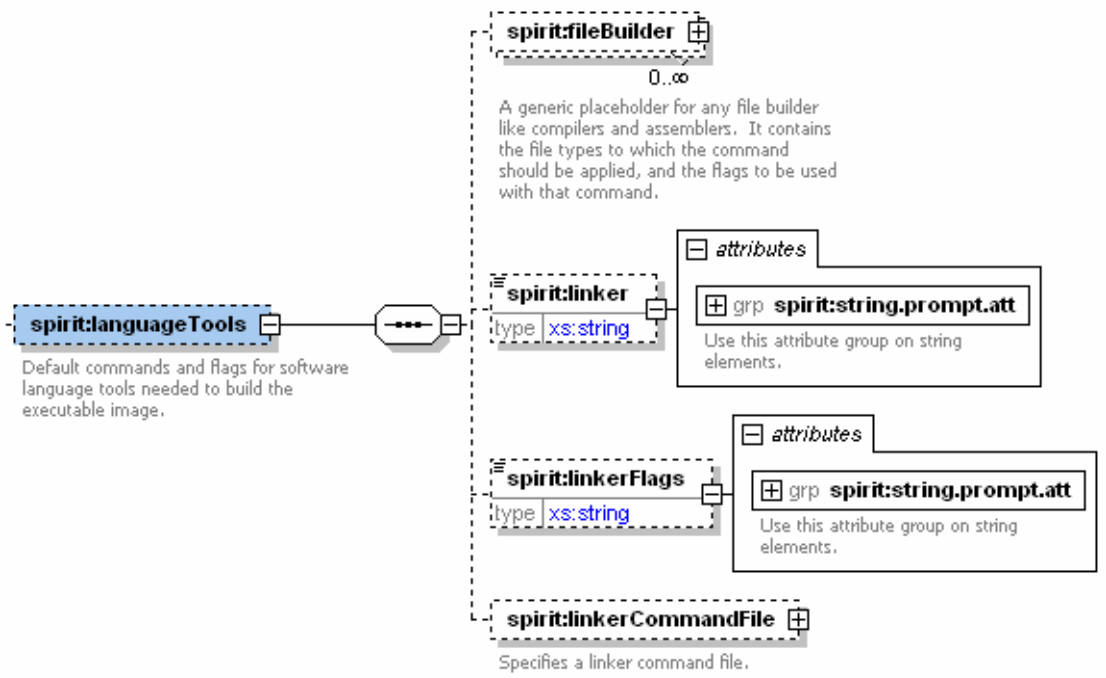
The following example shows the definition of a binary executable produced using the Gnu C Compiler (GCC) software tools.

```
<spirit:executableImage spirit:id="gnu" spirit:imageType="bin">
   <spirit:name>calculator.x</spirit:name>
   <spirit:description>Calculator function</spirit:name>
   <spirit:languageTools>
      <spirit:fileBuilder>
         <spirit:fileType>cSource</spirit:fileType>
         <spirit:command spirit:id="gccCompilerDefault"> gcc</
spirit:command>
         <spirit:flags spirit:id="gccCFlags">-c -g -I${INCLUDES_LOCATION}/
software/include -I${GCC_LIBRARY}/common/include</spirit:flags>
      </spirit:fileBuilder>
      <spirit:fileBuilder>
         <spirit:fileType>asmSource</spirit:fileType>
         <spirit:command spirit:id="gccAssemblerDefault">gcc</
spirit:command>
         <spirit:flags spirit:id="gccAsmFlags">-c -Wa,--gdwarf2 -
I${INCLUDES_LOCATION}/software/include -I${GCC _LIBRARY}/common/include</
spirit:flags>
      </spirit:fileBuilder>
      <spirit:linker spirit:id="gccLinker">gcc</spirit:linker>
      <spirit:linkerFlags spirit:id="gccLnkFlags">-g -nostdlib -static -
mcpu=arm9</spirit:linkerFlags>
      <spirit:linkerCommandFile>
      <spirit:name spirit:id="lnkCmdFile">linker.ld</spirit:name>
      <spirit:commandLineSwitch spirit:id="lnkCmSwitch">-T</
spirit:commandLineSwitch>
      <spirit:enable spirit:id="lnkCmdEnable">true</spirit:enable>
      <spirit:generatorRef>org.spiritconsortium.tool</spirit:generatorRef>
      </spirit:linkerCommandFile>
   </spirit:languageTools>
   <spirit:fileSetRefGroup>
      <spirit:fileSetRef>calculatorAppC</spirit:fileSetRef>
      <spirit:fileSetRef>mathFunctions</spirit:fileSetRef>
      <spirit:fileSetRef>coreLib-gnu</spirit:fileSetRef>
   </spirit:fileSetRefGroup>
</spirit:executableImage>
```

7.7.3 languageTools

7.7.3.1 Schema

The following schema details the information contained in the **languageTools** element, which may appear as an element inside the **executableImage** element.



7.7.3.2 Description

The **languageTools** element contains the following list of optional elements to document a set of software tools used to create an executable binary documented by the parent **executableImage** element. Multiple **languageTools** information can be created to reflect various software tool sets that can create this executable binary file.

a) **fileBuilder** (optional) contains the information details of a compiler or assembler for software source code. See 7.7.4.

b) **linker** (optional) documents the link editor associated with the software tools described in **fileBuilder**. The **linker** element is of type string. The **linker** element is configurable with attributes from ***string.prompt.att***, see X.Y.Z on configuration.

c) **linkerFlags** (optional) can also be associated with any **linker** information. The **linkerFlags** element is of type string. The **linkerFlags** element is configurable with attributes from ***string.prompt.att***, see X.Y.Z on configuration.

d) **linkerCommandFile** (optional) documents a file containing commands the linker follows. See 7.7.5.

7.7.3.3 Example

The following example shows the definition of GCC software tools used together to produce an executable binary code file.

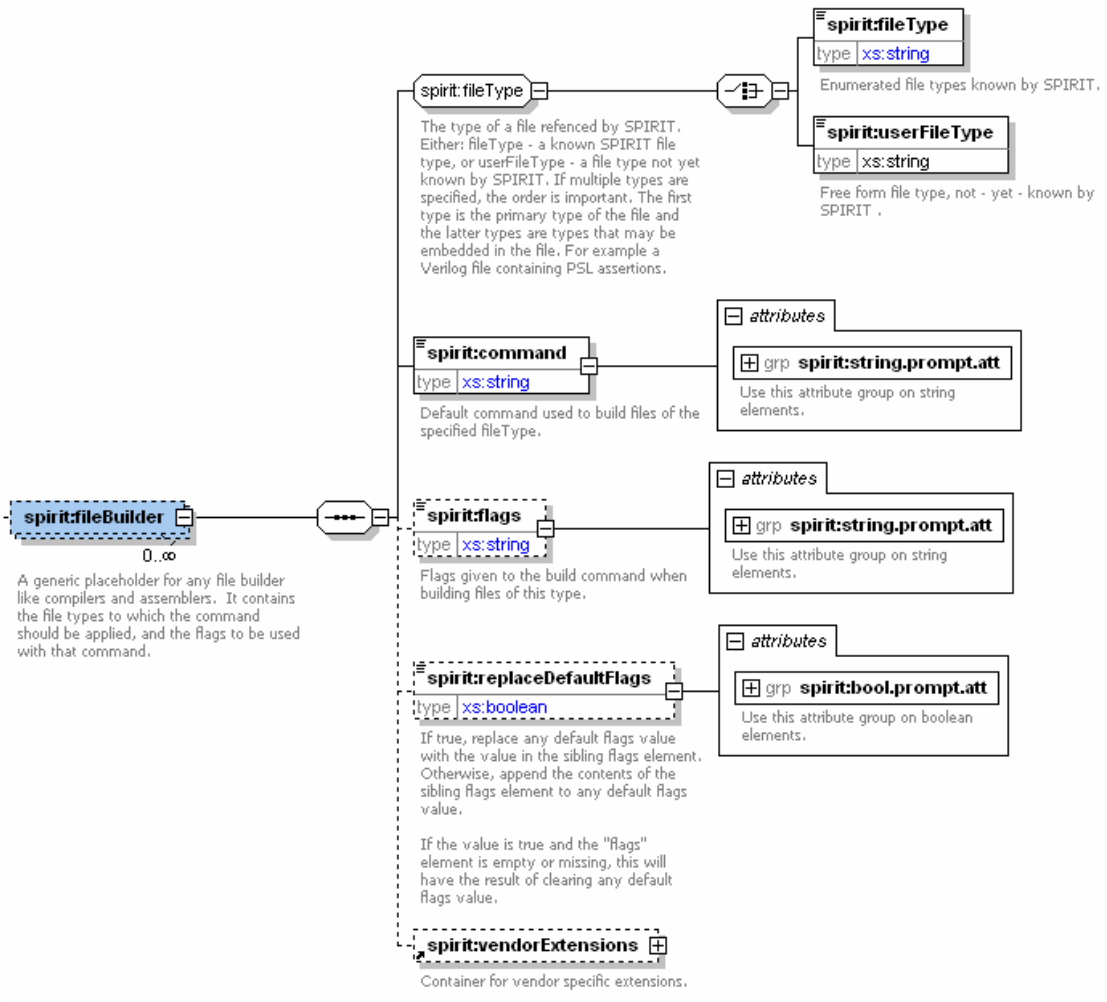
```
<spirit:languageTools>
    <spirit:fileBuilder>
        <spirit:fileType>cSource</spirit:fileType>
        <spirit:command spirit:id="gccCompilerDefault"> gcc</spirit:command>
        <spirit:flags spirit:id="gccCFlags">-c -g -I${INCLUDES_LOCATION}/
    software/include -I${GCC_LIBRARY}/common/include</spirit:flags>
    </spirit:fileBuilder>
```

```
        <spirit:fileBuilder>
            <spirit:fileType>asmSource</spirit:fileType>
            <spirit:command spirit:id="gccAssemblerDefault">gcc</spirit:command>
            <spirit:flags spirit:id="gccAsmFlags">-c -Wa,--gdwarf2 -
        I${INCLUDES_LOCATION}/software/include -I${GCC _LIBRARY}/common/include</
        spirit:flags>
        </spirit:fileBuilder>
        <spirit:linker spirit:id="gccLinker">gcc</spirit:linker>
        <spirit:linkerFlags spirit:id="gccLnkFlags">-g -nostdlib -static -
        mcpu=arm9</spirit:linkerFlags>
        <spirit:linkerCommandFile>
            <spirit:name spirit:id="lnkCmdFile">linker.ld</spirit:name>
            <spirit:commandLineSwitch spirit:id="lnkCmSwitch">-T</
        spirit:commandLineSwitch>
            <spirit:enable spirit:id="lnkCmdEnable">true</spirit:enable>
            spirit:generatorRef>org.spiritconsortium.tool</spirit:generatorRef>
        </spirit:linkerCommandFile>
    </spirit:languageTools>
```

7.7.4 fileBuilder

7.7.4.1 Schema

The following schema details the information contained in the **fileBuilder** element, which may appear as an element inside a **languageTools** element within the **executableImage** element.



7.7.4.2 Description

The **fileBuilder** element contains the following mandatory and optional elements.

a) ***fileType*** group includes the following, of which one is required.

fileType (required) describes a file containing software source code in a language type recognized by IP-XACT, see XXX for a list of valid choices; otherwise, **userFileType** (required) can be used to specify any user-defined language type.

b) **command** (optional) element defines a compiler or assembler tool that processes the software of this type. The **command** element is of type ***string***. The **command** element is configurable with attributes from ***string.prompt.att***, see X.Y.Z on configuration.

c) **flags** (optional) documents any flags to be passed along with the software tool command. The **flags** element is of type ***string***. The **flags** element is configurable with attributes from ***string.prompt.att***, X.Y.Z on configuration.

d) **replaceDefaultFlags** (optional) documents flags that replace any of the passed default flags. The **replaceDefaultFlags** element is of type ***Boolean***. The **replaceDefaultFlags** element is configurable with attributes from ***bool.prompt.att***, see X.Y.Z on configuration.

e) **vendorExtensions** (optional) holds vendor-specific data from other name spaces applicable to building this software source code file into an executable object file.

7.7.4.3 Example

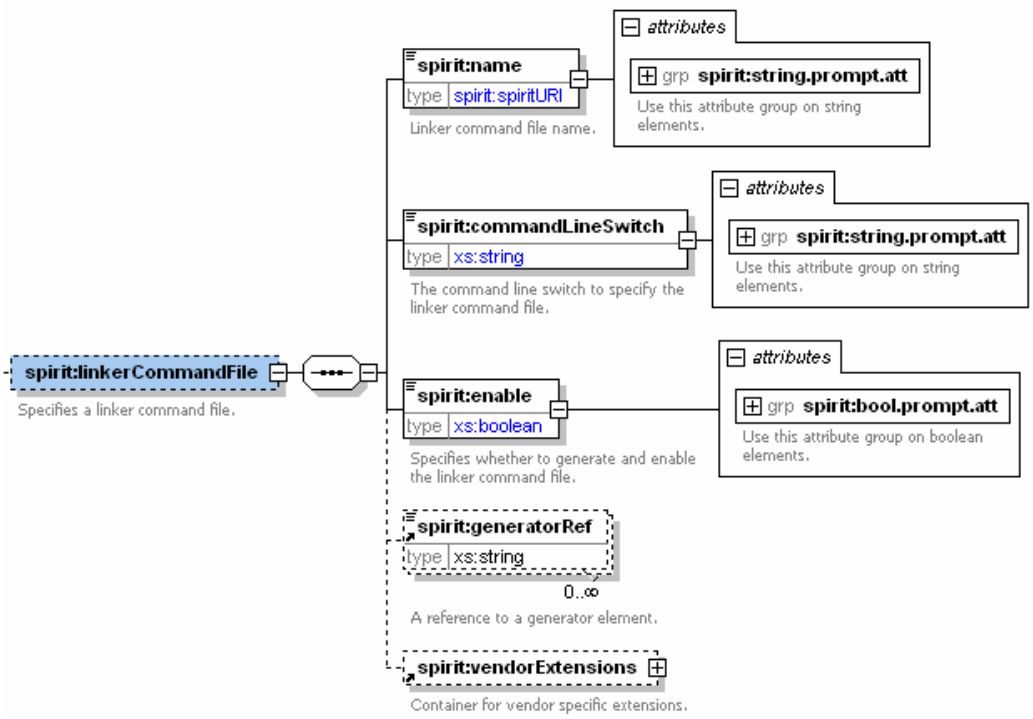
The following example shows the specification for compiling a C language file using GCC.

```
<spirit:fileBuilder>
   <spirit:fileType>cSource</spirit:fileType>
   <spirit:command spirit:id="gccCompilerDefault"> gcc</spirit:command>
   <spirit:flags spirit:id="gccCFlags">-c -g -I${INCLUDES_LOCATION}/software/
   include -I${GCC_LIBRARY}/common/include</spirit:flags>
</spirit:fileBuilder>
```

7.7.5 linkerCommandFile

7.7.5.1 Schema

The following schema details the information contained in the **linkerCommandFile** element, which may appear as an element inside a **languageTools** element within the **executableImage** element.



7.7.5.2 Description

The **linkerCommandFile** element contains information related to contents of the **linker** and **linkerFlags** elements, specifically about a file containing linker commands. It contains the following mandatory and optional elements.

a) **name** (mandatory) documents the location and name of the file containing commands for the linker. The **name** element is of type spiritURI. The **name** element is configurable with attributes from ***string.prompt.att,*** see X.Y.Z on configuration.

b) **commandLineSwitch** (mandatory) documents the flag on the command line calling the linker. The **commandLineSwitch** element is of type spiritURI. The **commandLineSwitch** element is configurable with attributes from ***string.prompt.att,*** see X.Y.Z on configuration.

c) **enable** (mandatory) indicates whether to use this linker command file in the default scenario. The **enable** element is of type Boolean. The **enable** element is configurable with attributes from ***bool.prompt.att,*** see X.Y.Z on configuration.

d) **generatorRef** (optional) documents the generator that creates and launches the linker command. There may be any number of these elements present.

e) **vendorExtensions** (optional) holds any vendor-specific data from other name spaces applicable to using this linker.

7.7.5.3 Example

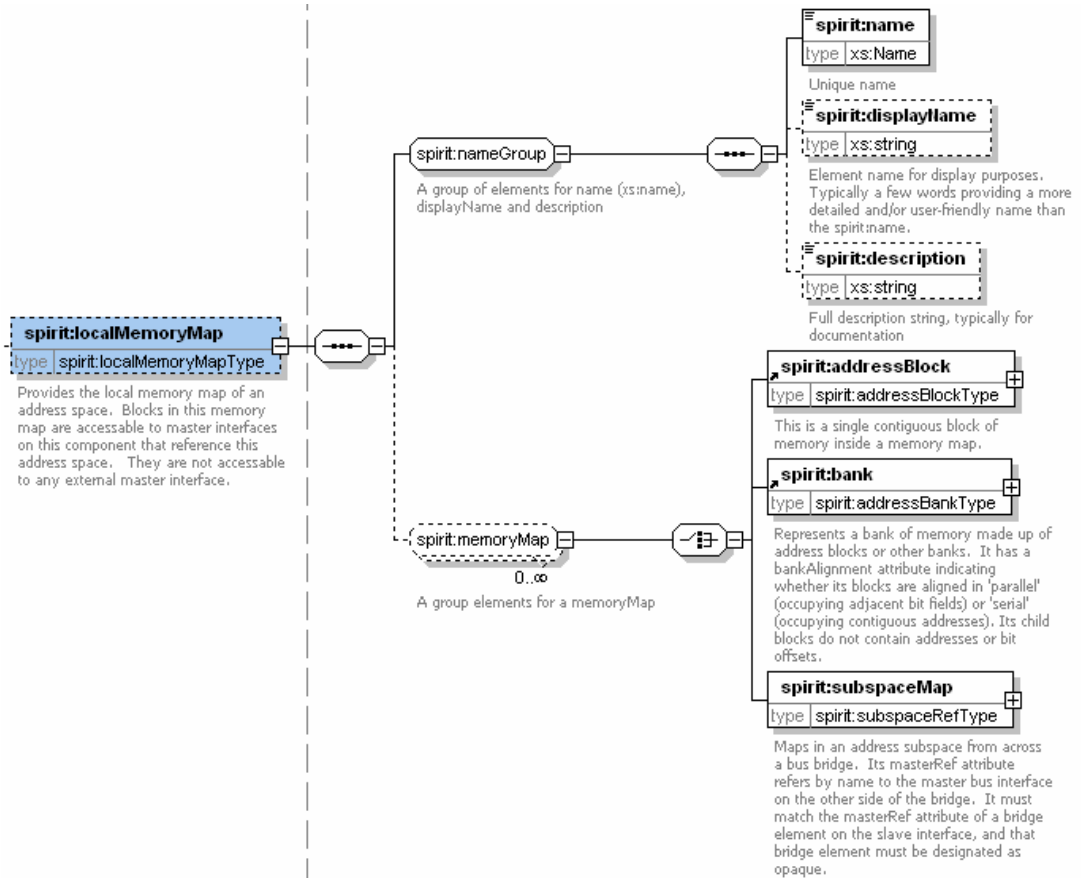
The following example shows the definition of a status register which can be accessed within a component during verification.

```
<spirit:linkerCommandFile>
    <spirit:name spirit:id="linkerCommandFileName2">linker.ld</spirit:name>
    <spirit:commandLineSwitch spirit:id="lnkCmSwitch">-T</
    spirit:commandLineSwitch>
    <spirit:enable spirit:id="lnkCmdEnable">true</spirit:enable>
    <spirit:generatorRef>org.spiritconsortium.tool.gccLinkerLauncher</
    spirit:generatorRef>
</spirit:linkerCommandFile>
```

7.7.6 Local memory map

7.7.6.1 Schema

The following schema details the information contained in the **localMemoryMap** element, which may appear inside an **addressSpace** element.



7.7.6.2 Description

Some processor components require specifying a memory map that is local to the component. *Local memory maps* (the **localMemoryMap** element in the **addressSpace** element of the component) are blocks of memory within a component that can only be accessed by the master interfaces of that component.

a) ***nameGroup*** group includes the following. See X.Y.Z.
   1) **name** (mandatory) identifies the address space.
   2) **displayName** (optional) allows a short descriptive text to be associated with the address space.
   3) **description** (optional) allows a textual description of the address space.

b) ***memoryMap*** group (optional) is any number of the following.
   1) **addressBlock** describes a single block. See 7.8.2.
   2) **bank** represents a collections of address blocks, banks or subspace maps. See 7.8.4.
   3) **subspaceMap** maps the address subspaces of master interfaces into the slave's memory map. See 7.8.8.

7.7.6.3 Example

The following example shows a secure register space with limited access to the master bus interface as the definition of a local memory map for an address space.

```
<spirit:localMemoryMap>
   <spirit:name>secureRegs</spirit:name>
```

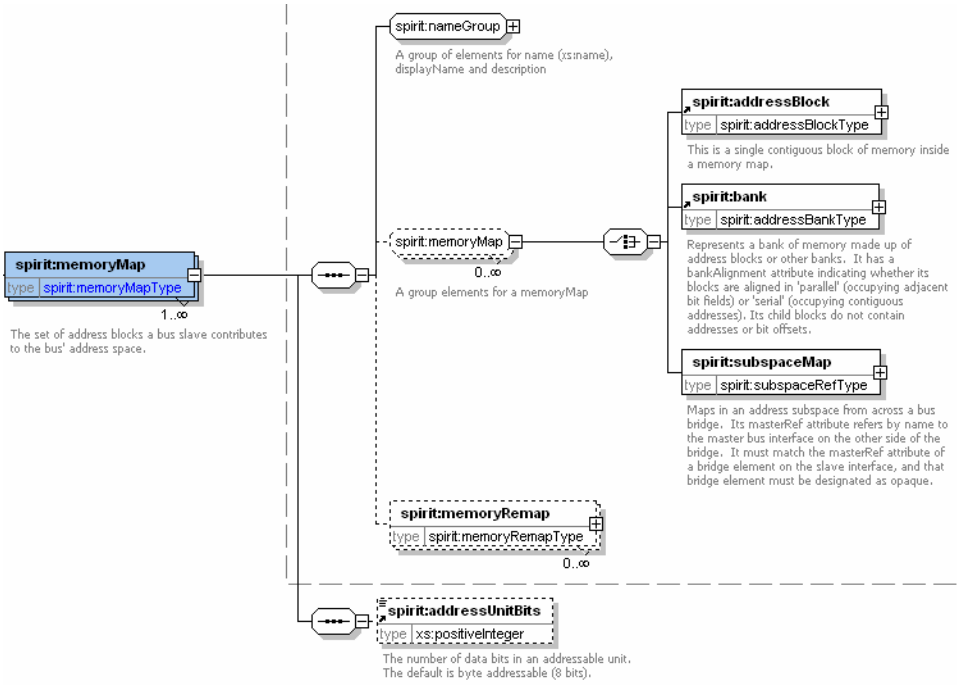
```
    <spirit:displayName>Secure Registers</spirit:displayName>
    <spirit:description>Secure registers area</spirit: description>
    <spirit:addressBlock>
       <spirit:baseAddress spirit:id="secureRegs">0x50000000</
    spirit:baseAddress>
       <spirit:range>64</spirit:range>
       <spirit:width>32</spirit:width>
       <spirit:usage>register</spirit:usage>
       <spirit:access>read-write</spirit:access>
    </spirit:addressBlock>
</spirit:localMemoryMap>
```

7.8 Memory maps

7.8.1 Memory map

7.8.1.1 Schema

The following schema details the information contained in the **memoryMap** element, which may appear as an element inside the **component** element. It is of type ***memoryMapType***.



7.8.1.2 Description

A memory map can be defined for each slave interface of a component. The **memoryMap** element is defined at the top of the component and then referenced in a component slave interface. It contains the following mandatory and optional elements.

a) ***nameGroup*** group includes the following. See X.Y.Z .

   1) **name** (mandatory) identifies the memory map.

   2) **displayName** (optional) allows a short descriptive text to be associated with the memory map.

   3) **description** (optional) allows a textual description of the memory map.

b) ***memoryMap*** group (optional) is any number of the following.

   1) **addressBlock** describes a single block. See 7.8.2.

   2) **bank** represents a collections of address blocks, banks or subspace maps. See 7.8.4.

   3) **subspaceMap** maps the address subspaces of master interfaces into the slave's memory map. See 7.8.8.

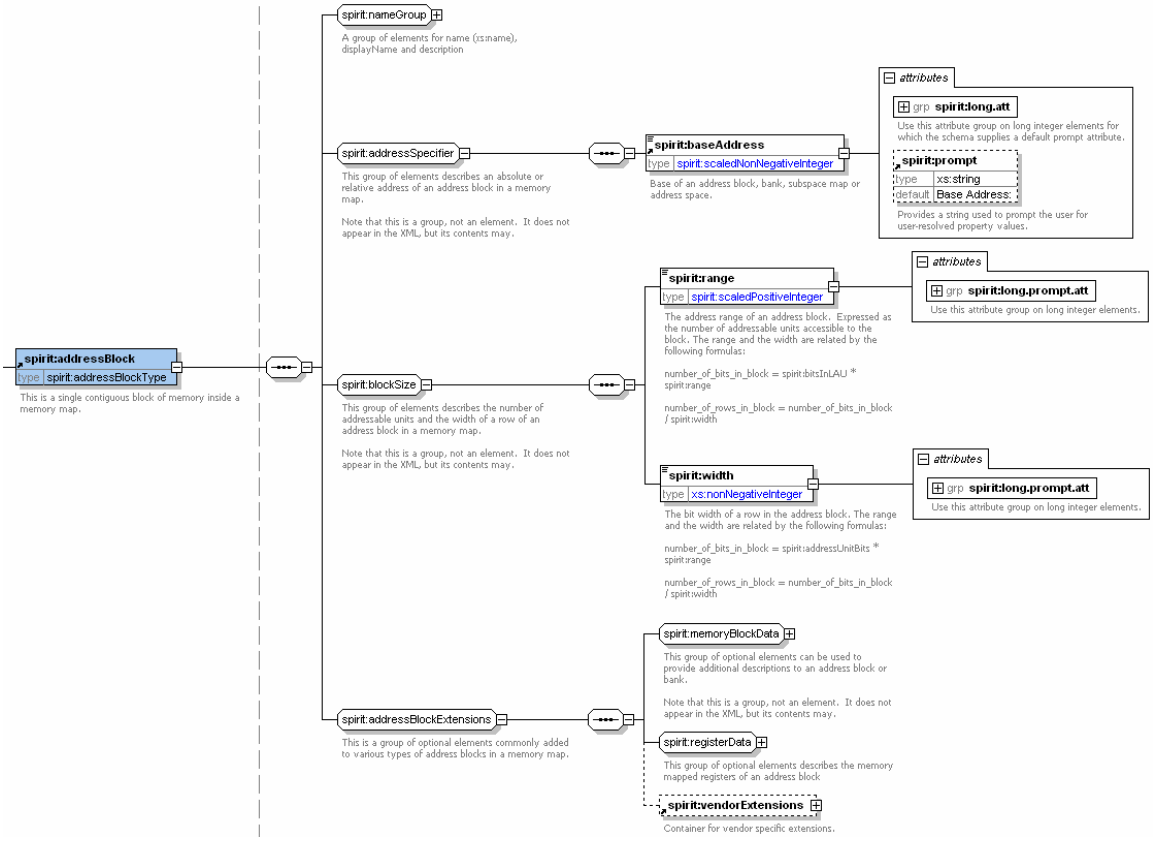
c) The optional **memoryRemap** element describes how the address spaces, banks and subspace maps are to be mapped differently on a slave bus interface in a specific remap **state**.

d) The optional **addressUnitBits** elements defines the number of data bits in each address increment of the memory map. This is required to allow the elements in the memory map to define items such as register offsets.

7.8.2 Address block

7.8.2.1 Schema

The following schema details the information contained in the **addressBlock** element, which may appear in a **memoryMap** element. It is of type ***addressBlockType***.



7.8.2.2 Description

The **addressBlock** element describes a single, contiguous block of memory that is part of a memory map. It contains the following mandatory and optional elements.

a) ***nameGroup*** group includes the following. See X.Y.Z.

1) **name** (mandatory) identifies the address block.

2) **displayName** (optional) allows a short descriptive text to be associated with the address block.

3) **description** (optional) allows a textual description of the address block.

b) ***addressSpecifier*** group includes the following.

1) **baseAddress** (mandatory) specifies the starting address of the block. The **baseAddress** element is of type ***scaledNonNegativeInteger***. The **baseAddress** element is configurable with

attributes from ***long.att***, see X.Y.Z on configuration. The **prompt** (optional) attribute allow the setting of a string for the configuration and has a default value of "Base Address:".

c) ***blockSize*** group includes the following.

2) **range** (mandatory) gives the address range of an address block. This is expressed as the number of addressable units of the memory map. The size of an addressable unit is defined inside the containing **memoryMap/addressUnitBits** element. The **range** element is of type ***scaledPositiveInteger***. The **range** element is configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

3) **width** (mandatory) is the bit width of a row in the address block. A row in an address block sets the maximum single transfer size into the memory map allowed by the referencing bus interface and also defines the maximum size that a single register can be defined across an interconnection. The **width** element is of type ***nonNegativeInteger***. The **width** element is configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

d) ***memoryBlockData*** group contains information about usage, access, volatility and other parameters. See 7.8.3.

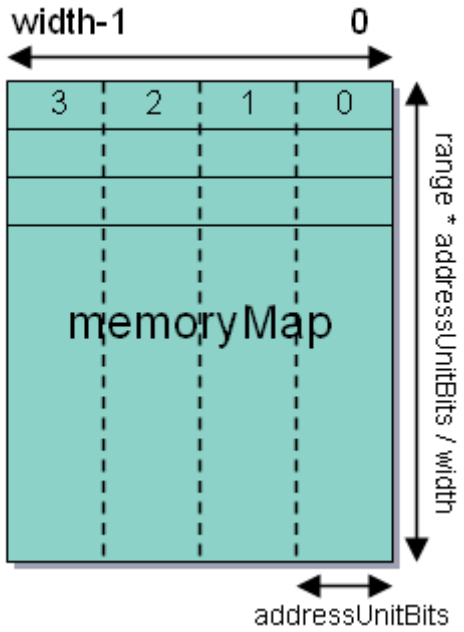
e) ***registerData*** group contains information about the grouping of bits into registers and fields. See 7.10.1.

f) **vendorExtensions** (optional) adds any extra vendor-specific data related to the address block.

The **range** and **width** elements are related by the following formulas

number_of_bits_in_block = `addressUnitBits` * `range`

number_of_rows_in_block = *number_of_bits_in_block* / `width`



See also: SCR 8.1.

7.8.2.3 Example

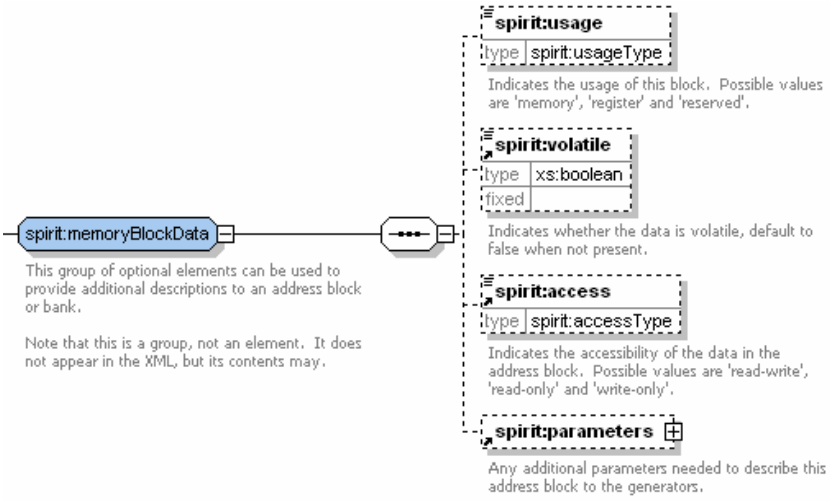
The following example shows an address block starting at address `0x1000` containing 64 addressable 8 -bit units, organized into larger 32-bit units.

```
<spirit:memoryMap>
<spirit:addressBlock>
  <spirit:name>AB1</spirit:name>
  <spirit:baseAddress>0x1000</spirit:baseAddress>
  <spirit:range>64</spirit:range>
  <spirit:width>32</spirit:width>
</spirit:addressBlock>
<spirit:addressUnitBits>8</spirit:addressUnitBits>
</spirit:memoryMap>
```

7.8.3 memoryBlockData group

7.8.3.1 Schema

The following schema details the information contained in the ***memoryBlockData*** group, an optional part of both **addressBlock** and **bank**.



7.8.3.2 Description

The ***memoryBlockData*** group is a collection of elements that contains further specification of **addressBlock** or **bank** elements. It contains the following optional elements. (needs data from Gary added)

a) **usage** (optional) specifies the type of usage for the block or bank to which it belongs: ***memory***, ***register***, or ***reserved***.

b) **volatile** (optional) is of type Boolean and indicates the data is volatile when set to ***True***. The default is ***False***.

c) **access** (optional) specifies the accessibility of the data in the address block: ***read-write***, ***read-only***, or ***write-only***.

d) **parameters** (optional) details any additional parameters that describe the address block for generator usage. See X.Y.Z.

7.8.3.3 Example

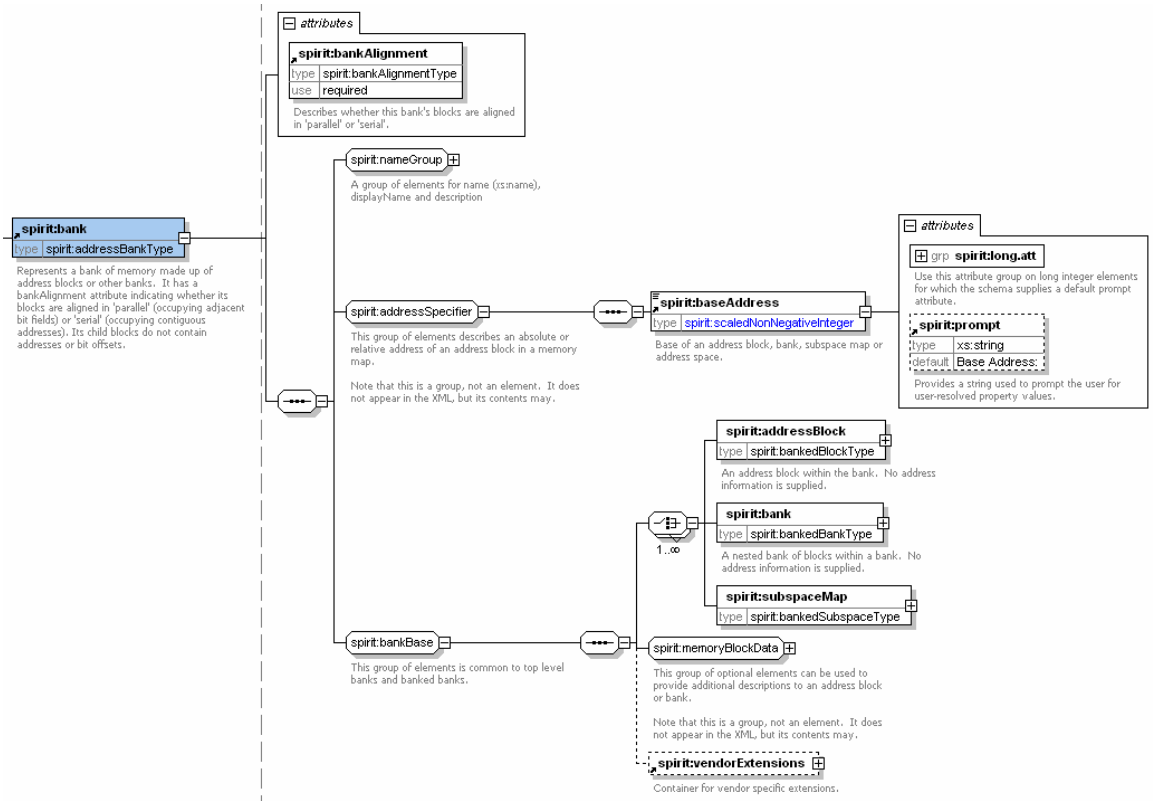
The following example shows an address block starting at address `0x0` containing 64 addressable memory locations of 8 bits, organized into larger 32-bit units.

```
<spirit:memoryMap>
<spirit:addressBlock>
  <spirit:name>AB1</spirit:name>
  <spirit:baseAddress>0</spirit:baseAddress>
  <spirit:range>64</spirit:range>
  <spirit:width>32</spirit:width>
  <spirit:usage>memory</spirit:width>
<spirit:volatile>false</spirit:volatile>
<spirit:access>read-write</spirit:access>
</spirit:addressBlock>
<spirit:addressUnitBits>8</spirit:addressUnitBits>
</spirit:memoryMap>
```

7.8.4 Bank

7.8.4.1 Schema

The following schema details the information contained in the **bank** element, which can appear in a **memoryMap** element. It is of type ***addressBankType***.



7.8.4.2 Description

The **bank** element allows multiple address blocks, banks or subspaceMaps to be concatenated together horizontily or veritically a single entity. It contains the following attributes and elements.

a) **bankAlignment** (mandatory) attribute organizes the bank:

   1) ***parallel*** specifies each item is located at the same base address with different bit offsets. The bit offset of the first item in the bank always starts at 0, the offset of the next items in the bank is equal to the widths of all the previous items.

   2) ***serial*** specifies the first item is located at the bank's base address. Each subsequent item is located at the previous item's address, plus the range of that item(adjusted for LAU and bus width considerations, rounded up to the next whole multiple). This allows the user to specify only a single base address for the bank and have each item line up correctly.

b) ***nameGroup*** group includes the following. See X.Y.Z.

   1) **name** (mandatory) identifies the bank.

   2) **displayName** (optional) allows a short descriptive text to be associated with the bank.

   3) **description** (optional) allows a textual description of the bank.

c) ***addressSpecifier*** group includes the following.

   1) **baseAddress** (mandatory) specifies the starting address of the block. The type of this element is set to scaledNonNegativeInteger. The **baseAddress** element is configurable with attributes from ***bool.prompt.att***, see X.Y.Z on configuration. The **prompt** attribute allow the setting of a string for the configuration and has a default value of "Base Address:".

d) ***bankBase*** group include the following. This group is later used inside the ***bankedBaseType*** type to create recursion.

   1) **addressBlock** (multiple usage allowed) is an address block that makes up part of the bank. See 7.8.5.

   2) **bank** (multiple usage allowed) is a bank within the bank. This allows for complex configurations with nested banks. See 7.8.6.

   3) **subspaceMap** (multiple usage allowed) is a reference to the master's address map for inclusion in the bank. See 7.8.8.

   4) ***memoryBlockData*** group contains information about usage, access, volatility and other parameters. See 7.8.3.

   5) **vendorExtensions** adds any extra vendor-specific data related to this bank.

See also: SCR 8.2 and SCR 8.3.

7.8.4.3 Example

The following example shows a serial bank with four memory blocks of 1K units of 8-bit data. The only address specified is `0x10000`, but this causes address block `ram0`, `ram1`, `ram2`, and `ram3` to be mapped to addresses `0x10000`, `0x11000`, `0x11000 0x12000`, and `0x13000` respectively.

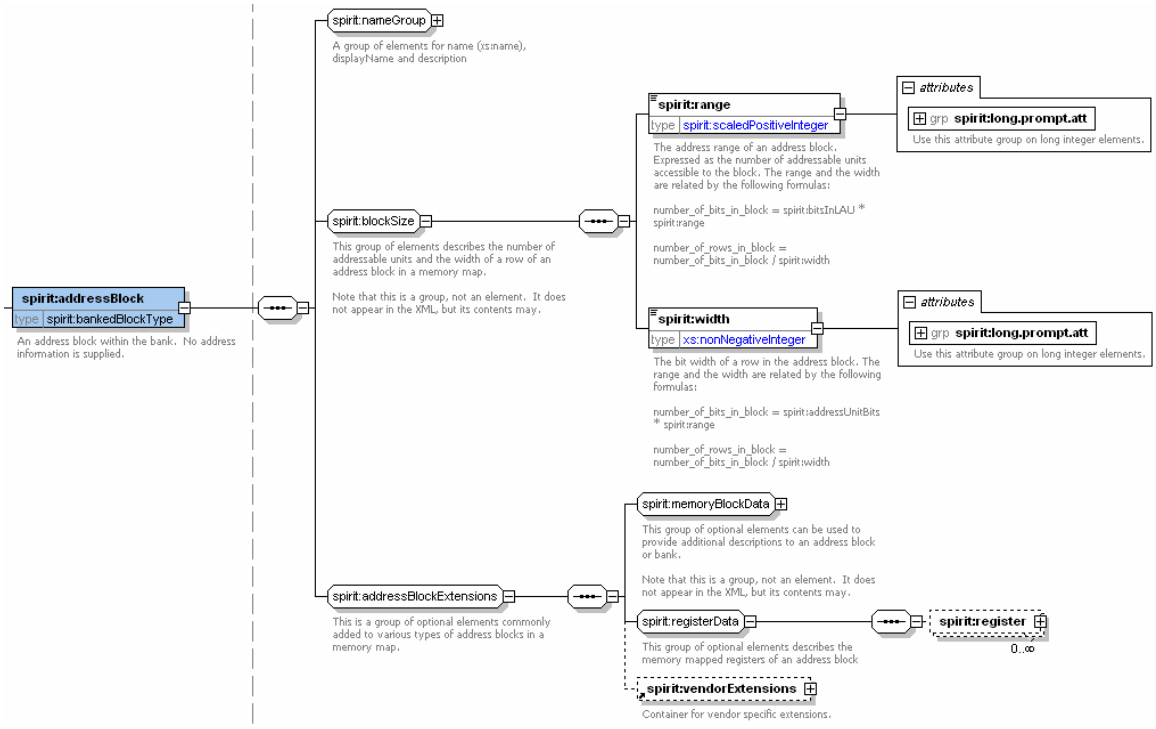
```
<spirit:memoryMap>
<spirit:bank bankAlignment="serial">
    <spirit:name>bank1</spirit:name>
    <spirit:baseAddress>0x10000</spirit:baseAddress>
    <spirit:addressBlock>
      <spirit:name>ram0</spirit:name>
      <spirit:range>0x1000</spirit:range>
      <spirit:width>32</spirit:width>
    </spirit:addressBlock>
    <spirit:addressBlock>
```

```
        <spirit:name>ram1</spirit:name>
        <spirit:range>0x1000</spirit:range>
        <spirit:width>32</spirit:width>
      </spirit:addressBlock>
      <spirit:addressBlock>
        <spirit:name>ram2</spirit:name>
        <spirit:range>0x1000</spirit:range>
        <spirit:width>32</spirit:width>
      </spirit:addressBlock>
      <spirit:addressBlock>
        <spirit:name>ram3</spirit:name>
        <spirit:range>0x1000</spirit:range>
        <spirit:width>32</spirit:width>
      </spirit:addressBlock>
  </spirit:bank>
  <spirit:addressUnitBits>8</spirit:addressUnitBits>
  </spirit:memoryMap>
```

7.8.5 Banked address block

7.8.5.1 Schema

The following schema details the information contained in the **addressBlock** element, which can appear in a **bank** element. It is of type ***bankedBlockType***.



7.8.5.2 Description

The **addressBlock** element inside a bank element describes a single, contiguous block of memory that is part of a bank. It contains the following elements.

a) ***nameGroup*** group includes the following. See X.Y.Z.

1) **name** (mandatory) identifies the address block.
2) **displayName** (optional) allows a short descriptive text to be associated with the address block.
3) **description** (optional) allows a textual description of the address block.

b) ***blockSize*** group includes the following.

4) **range** (mandatory) gives the address range of an address block. This is expressed as the number of addressable units of the memory map. The size of an addressable unit is defined inside the containing **memoryMap/addressUnitBits** element. The type of this element is set to scaledPositiveInteger. The **range** element is configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.
5) **width** (mandatory) is the bit width of a row in the address block. A row in an address block sets the maximum single transfer size into the memory map allowed by the referencing bus interface and also defines the maximum size that a single register can be defined across. The type of this element is set to nonNegativeInteger. The **width** element is configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

c) ***memoryBlockData*** group contains information about usage, access, volatility and other parameters. See 7.8.3.

d) ***registerData*** group contains information about the grouping of bits into registers and fields. See 7.10.1.

e) **vendorExtensions** (optional) adds any extra vendor-specific data related to the address block.

NOTE—The ***bankedBlockType*** of a **addressBlock** element is almost identical to the ***addressBlockType*** of an **addressBlock** element(see 7.8.2); the only difference is there is no **baseAddress** in the ***bankedBlockType*** version.

See also: SCR 8.3.

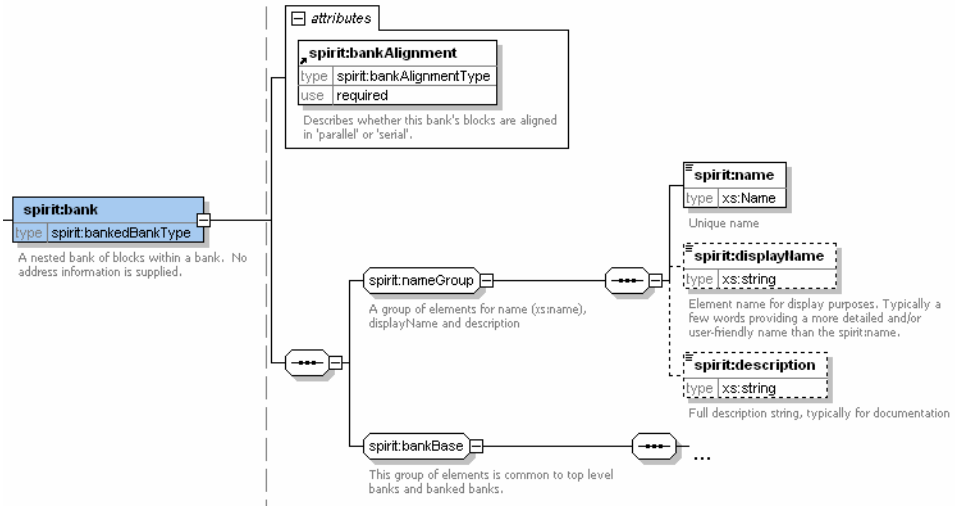
7.8.5.3 Example

See the example in 7.8.4.3.

7.8.6 Banked bank

7.8.6.1 Schema

The following schema details the information contained in the nested **bank** element, which can appear in another **bank** element. It is of type ***bankBankType***.



7.8.6.2 Description

The **bank** element allows multiple address blocks, banks or subspace maps to be to be concatenated together horizontily or veritically a single entity. It contains the following attributes and elements.

a) **bankAlignment** (mandatory) attribute organizes the bank:

   1) ***parallel*** specifies each item is located at the same base address with different bit offsets. The bit offset of the first item in the bank always starts at 0, the offset of the next items in the bank is equal to the widths of all the previous items.

   2) ***serial*** specifies the first item is located at the bank's base address. Each subsequent item is located at the previous item's address, plus the range of that item(adjusted for LAU and bus width considerations, rounded up to the next whole multiple). This allows the user to specify only a single base address for the bank and have each item line up correctly.

b) ***nameGroup*** group includes the following. See X.Y.Z.

   1) **name** (mandatory) identifies the bank.

   2) **displayName** (optional) allows a short descriptive text to be associated with the bank.

   3) **description** (optional) allows a textual description of the bank.

c) The **bank** element or type ***bankedBankType*** then contains the ***bankBase*** group. This group is defined inside the **bank** element of type ***addressBankType***. See X.Y.Z. The effect of its inclusion here creates recursion, where by banks maybe included inside banks included inside banks.

NOTE—A banked bank is similar to a bank in a memory map (see 7.8.4); the only difference is there is no **baseAddress** element in a **bank** of type ***bankedBankType***.

See also: SCR 8.2 and SCR 8.3.

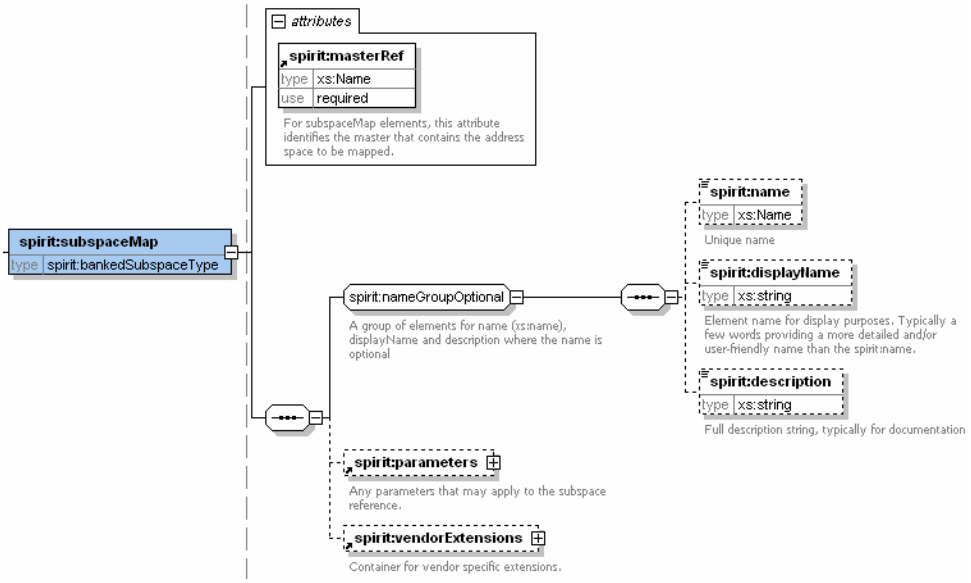
7.8.6.3 Example

Need example.

7.8.7 Banked subspace

7.8.7.1 Schema

The following schema details the information contained in the **subspaceMap** element, which can appear in a **bank** element. It is of type ***bankSubspaceType***.



7.8.7.2 Description

The **subspaceMap** element allows a bank to map the address space of a master interface into the bank. It contains the following elements.

a) **masterRef** attribute contains the name of the master interface whose address space needs to be mapped. This shall reference a bus interface name with a interface mode of master. The master interface must also be referenced by a second interface through a **slave/bridge/masterRef** element, and the **bridge** element shall also have the **opaque** attribute set to ***True***.

b) ***nameGroupOptional*** group includes the following. See X.Y.Z .

   1) **name** (optional) identifies the subspace map.

   2) **displayName** (optional) allows a short descriptive text to be associated with the subspace map.

   3) **description** (optional) allows a textual description of the subspace map.

c) **parameters** details any additional parameters that apply to the **subspaceMap**. See X.Y.Z.

d) **vendorExtensions** adds any extra vendor-specific data related to the **subspaceMap**.

See also: SCR 8.2.

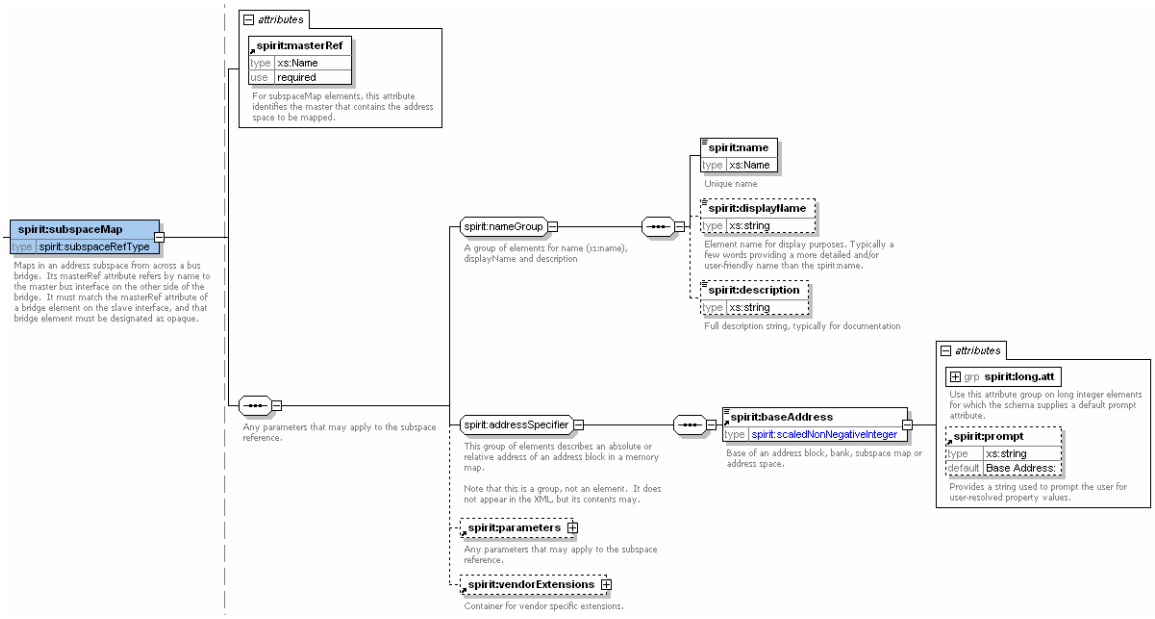
7.8.7.3 Example

Add an example here.

7.8.8 Subspace map

7.8.8.1 Schema

The following schema details the information contained in the **subspaceMap** element, which can appear in a **memoryMap** element. It is of type ***subspaceRefType***.



7.8.8.2 Description

The **subspaceMap** element maps the address subspace of a master interface from an opaque bus bridge into the memory map. It contains the following elements.

a) **masterRef** attribute contains the name of the master interface whose address space needs to be mapped. This shall reference a bus interface name with a interface mode of master. The master interface must also be referenced by a second interface through a **slave/bridge/masterRef** element, and the **bridge** element shall also have the **opaque** attribute set to ***True***.

b) ***nameGroup*** group includes the following. See X.Y.Z.

   1) **name** identifies the subspace map.

   2) **displayName** (optional) allows a short descriptive text to be associated with the subspace map.

   3) **description** (optional) allows a textual description of the subspace map.

c) ***addressSpecifier*** group includes the following.

   1) **baseAddress** (mandatory) specifies the starting address of the block. The type of this element is set to scaledNonNegativeInteger. The **baseAddress** element is configurable with attributes from ***long.att***, see X.Y.Z on configuration. The **prompt** attribute allow the setting of a string for the configuration and has a default value of “Base Address:”.

d) **parameters** details any additional parameters that apply to the **subspaceMap**. See X.Y.Z.

e) **vendorExtensions** adds any extra vendor-specific data related to the **subspaceMap**.

7.8.8.3 Example

The following example shows an address block starting at address `0x1000` containing 64 addressable 32-bit units.

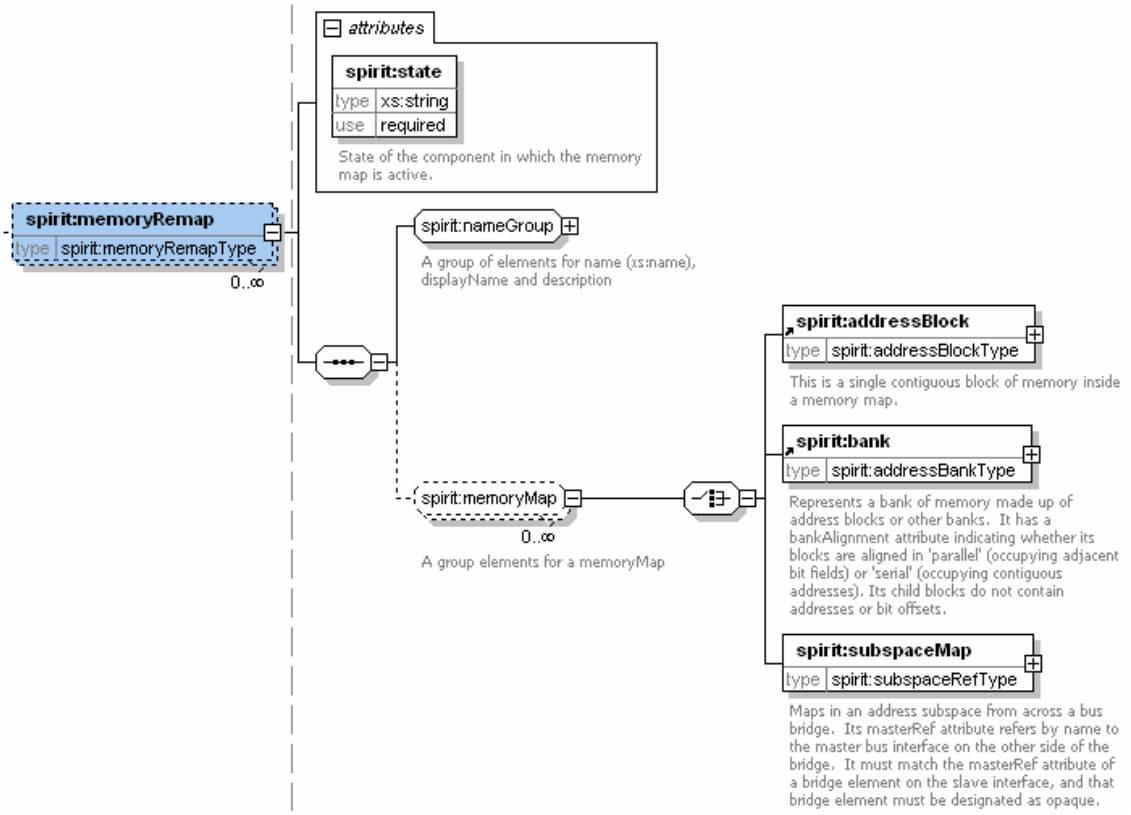
```
<spirit:component>…
  <spirit:busInterfaces>
    <spirit:busInterface>
      <spirit:name>M1</spirit:name>…
      <spirit:master>…</spirit:master>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>M2</spirit:name>?
      <spirit:master>…</spirit:master>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>M3</spirit:name>?
      <spirit:master>…</spirit:master>
    </spirit:busInterface>
    <spirit:busInterface>
      <spirit:name>S</spirit:name>?
      <spirit:slave>
        <spirit:memoryMapRef spirit:memoryMapRef="memMap"/>
        <spirit:bridge spirit:masterRef="M1" spirit:opaque="true"/>
        <spirit:bridge spirit:masterRef="M2" spirit:opaque="true"/>
        <spirit:bridge spirit:masterRef="M3" spirit:opaque="true"/>
      </spirit:slave>
    </spirit:busInterface>
  </spirit:busInterfaces>
  <spirit:addressSpaces>
    …
  </spirit:addressSpaces>
  <spirit:memoryMaps>
    <spirit:memoryMap>
      <spirit:name>memMap</spirit:name>
      <spirit:subspaceMap spirit:masterRef="M1">
        <spirit:name>submap1</spirit:name>
        <spirit:baseAddr baseAddress>0x0000</spirit:baseAddress>
      </spirit:subspaceMap>
      <spirit:subspaceMap spirit:masterRef="M2">
        <spirit:name>submap2</spirit:name>
        <spirit:baseAddress>0x1000</spirit:baseAddress>
      </spirit:subspaceMap>
      <spirit:subspaceMap spirit:masterRef="M3">
        <spirit:name>submap3</spirit:name>
        <spirit:baseAddress>0x2000</spirit:baseAddress>
      </spirit:subspaceMap>
    </spirit:memoryMap>
  </spirit:memoryMaps>
</spirit:component>
```

7.9 Remapping

7.9.1 Memory remap

7.9.1.1 Schema

The following schema details the information contained in the **memoryRemap** element, which can appear in a **memoryMap** element. It is of type ***memoryRemapType***.



7.9.1.2 Description

The **memoryRemap** element describes how the address space blocks need to be mapped on a slave bus interface in a specific remap **state**. This element contains the following elements, attributes and groups.

a) **state** attribute (mandatory) identifies the remap state name for which the alternate memory map is active. See 7.9.2.

b) ***nameGroup*** group includes the following. See X.Y.Z.

   1) **name** (mandatory) identifies the memory remap.

   2) **displayName** (optional) allows a short descriptive text to be associated with the memory remap.

   3) **description** (optional) allows a textual description of the memory remap.

c) ***memoryMap*** group (optional) is any number of the following.

   1) **addressBlock** describes a single block. See 7.8.2.

   2) **bank** represents a collections of address blocks, banks or subspace maps. See 7.8.4.

3) **subspaceMap** maps the address subspaces of master interfaces into the slave's memory map. See 7.8.8.

memoryRemap describes remapping in regular components and opaque bridges. Remap states include the name, associated remap port, and remap value). The memory map itself is partitioned in one or more memory remaps, which describe the memory layout for a particular state.

The following semantic rules apply to the **state** attribute.

— If there are duplicate state attributes in different **memoryRemap** tags in the same **memoryMap** section, only the first occurrence shall be recognized. In other words, the state attribute values of **memoryRemap** shall be unique within a **memoryMap** section.

— If a component has no **remapStates** tag specified, then the **memoryMap** is assumed to be in the default state.

— If a component has **remapStates** specified, but no **memoryRemap**, the first state listed is synonymous with the default state and shall match the **memoryMap** tag with no **state** attribute.

7.9.1.3 Example

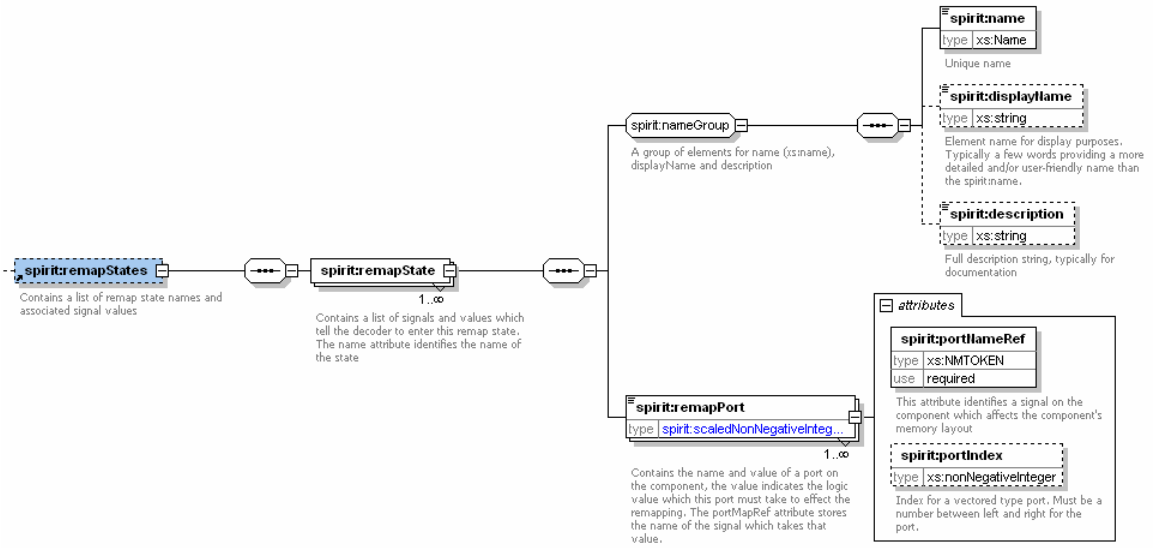
This is an example of a memory that is normally read-write, but in state lock is remapped to be a read-only memory.

```
<spirit:component>
…
  <spirit:memoryMaps>
    <spirit:memoryMap>
      <spirit:name>mmap1</spirit:name>
<spirit:memoryMap>
        <spirit:addressBlock>
        <spirit:name>ab1</spirit:name>
          <spirit:baseAddress>0x0000
          </spirit:baseAddress>
          <spirit:range>4096</spirit:range>
          <spirit:usage>memory</spirit:usage>
          <spirit:access>read-write</spirit:access>
        </spirit:addressBlock>
      </spirit:memoryRemap >
<spirit:memoryMap state="lock">
        <spirit:addressBlock>
        <spirit:name>ab1readonly</spirit:name>
          <spirit:baseAddress>0x0000
          </spirit:baseAddress>
          <spirit:range>4096</spirit:range>
          <spirit:usage>memory</spirit:usage>
          <spirit:access>read-only</spirit:access>
        </spirit:addressBlock>
      </spirit:memoryRemap >
    </spirit:memoryMap>
  </spirit:memoryMaps>
…
</spirit:component>
```

7.9.2 Remap states

7.9.2.1 Schema

The following schema details the information contained in the **remapStates** element, which may appear as an element inside a **component** element. This element may contain one or more **remapState** elements.



7.9.2.2 Description

A **remapStates** element describes a set of one or more **remapState** elements. Each **remapState** element defines a conditional remap state where each state is conditioned by a remap port specified with a **remapPort** element. A **remapState** element does not specify remapping addresses. The remapping addresses are defined by the **memoryRemap** element (of a **memoryMap** element) and its **state** attribute refers to the **remapState** element's name explained in this section.

remapState contains the following elements and attributes.

a) ***nameGroup*** group includes the following. See X.Y.Z.

   1) **name** (mandatory) identifies the memory remap.

   2) **displayName** (optional) allows a short descriptive text to be associated with the memory remap

   3) **description** (optional) allows a textual description of the memory remap.

b) **remapPort** specifies when the remap state gets effective. A collection of **remapPort** elements make up the condition for this remap state. All elements must be true for the remap state to be enabled. The type of this element is of ***scaledNonNegativeInteger***. This element contains the logical value of the single port bit specified by the follow two attributes.

   1) **portNameRef** (mandatory) attribute is the name of the port for which this logic value comparision is assigned.

   2) **portIndex** (optional) attribute references the index of a port when the port being referenced is vectored. The type of the attribute is nonNegativeInteger.

7.9.2.3 Example

This is an example of the **remapState** element with the state name of `boot`. The example specifies a remap state called `boot` will be in effect when the port named `doRemap` gets the logic value of `0x01`, while another remap state called `normal` will be in effect when the port gets the logic value of `0x00`.

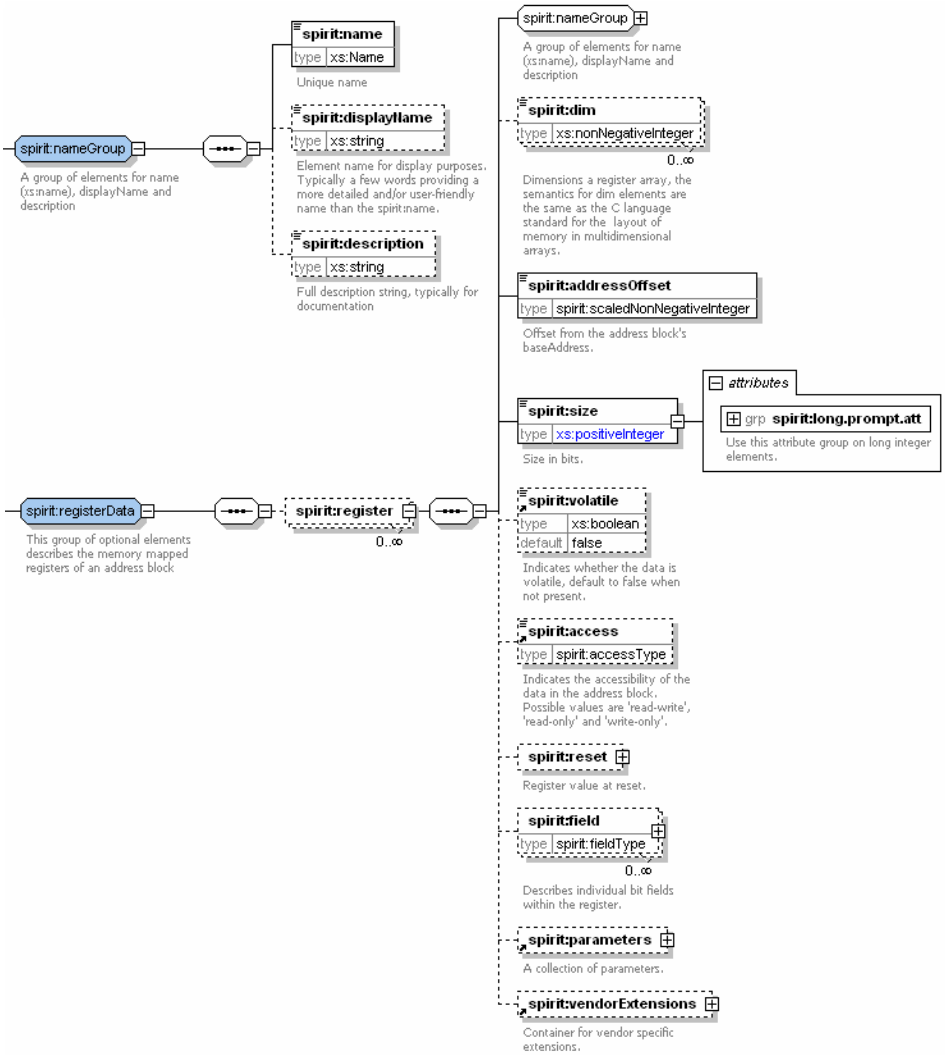
```
<spirit:component>
  <spirit:remapStates>
    <spirit:remapState>
      <spirit:name>boot</spirit:name>
      <spirit:remapPort spirit:portNameRef="doRemap">0x01
      </spirit:remapPort>
    </spirit:remapState>
    <spirit:remapState>
      <spirit:name>normal</spirit:name>
      <spirit:remapPort spirit:portNameRef="doRemap">0x00
      </spirit:remapPort>
    </spirit:remapState>
  </spirit:remapStates >
</spirit:component>
```

7.10 Registers

7.10.1 Register

7.10.1.1 Schema

The following schema details the information contained in the **register** element, which may appear as an element inside the **addressBlock** element. This element describes a register.



7.10.1.2 Description

the ***registerData*** group contains an unbounded list of **register** elements. A **register** element describes a register in an address block. This element contains the following elements.

a) ***nameGroup*** group includes the following. See X.Y.Z .
   1) **name** (mandatory) identifies the register.
   2) **displayName** (optional) allows a short descriptive text to be associated with the register.

3) **description** (optional) allows a textual description of the register.

b) **dim** (optional, unbounded) assigns a dimension to the register, so it is repeated as many times as the value of the **dim** elements. For multi-dimensional register arrays, the memory layout is presumed to follow the C language rules. The **dim** element is of type ***nonNegativeInteger***.

c) **addressOffset** describes the offset, in addressing units from the containing **memoryMap/ addressUnitBits** element. The offset is from the start of the **addressBlock**. The **addressOffset** element is of type ***scaledNonNegativeInteger.***

d) **size** (mandatory) is the width of the register, counting in bits. The type of this element is set to scaledNonNegativeInteger. The **size** element is configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

e) **volatile** (optional) if ***true*** indicates the data in the register is volatile; the default is ***false***. The type of this element is set to Boolean.

f) **access** (optional) indicates the accessibility of the register: ***read-write***, ***read-only***, or ***write-only***.

g) **reset** (optional) indicates the value of the register's contents when the device is reset. See 7.10.2.

h) **field** (optional) describes any bit-fields in a register. See 7.10.3.

i) **parameters** (optional) describes any parameter names and types when the register width can be parameterized.

j) **vendorExtensions** (optional) adds any extra vendor-specific data related to this register.

See also: SCR 7.1, SCR 7.2, SCR 7.3, and SCR 7.4.

7.10.1.3 Example

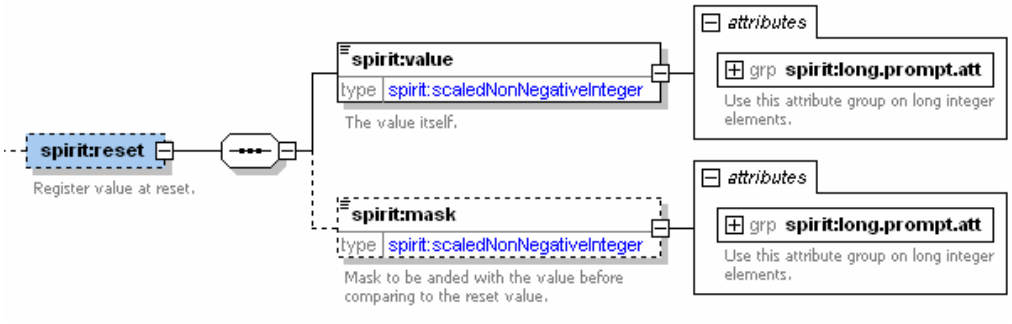
The following example shows a register with its sub-elements.

```
<spirit:register>
    <spirit:name>status</spirit:name>
    <spirit:description>Status register</spirit:description>
    <spirit:addressOffset>0x4</spirit:addressOffset>
    <spirit:size>32</spirit:size>
    <spirit:volatile>true</spirit:volatile>
    <spirit:access>read-only</spirit:access>
    <spirit:field>
        <spirit:name>dataReady</spirit:name>
        <spirit:description>Indicates that new data is available in the receiver
    holding register</spirit:description>
        <spirit:bitOffset>0</spirit:bitOffset>
        <spirit:bitWidth>1</spirit:bitWidth>
        <spirit:access>read-only</spirit:access>
    </spirit:field>
    <spirit:field>
    <!-- … -->
    </spirit:field>
</spirit:register>
```

7.10.2 Register reset value

7.10.2.1 Schema

The following schema details the information contained in the **reset** element, which may appear as an element inside the **register** element. This element describes the reset value of the register.



7.10.2.2 Description

The **reset** element describes the value of a register at reset. It has two subelements.

a) **value** (mandatory) contains the actual reset value. The **value** element is of type ***scaledNonNegativeInteger***. The **value** element is configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

b) **mask** (optional) defines which bits of the register have a known reset value. The **mask** element is of type ***scaledNonNegativeInteger***. The **mask** element is configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

   A `1` bit in the **mask** means the corresponding bit of the register has a known reset value; a `0` bit means that it does not. All bits of the **value** which correspond to `0` bits of the **mask** are ignored. The absence of a mask element is equivalent to a mask of the same size as the register consisting of all `1` bits.

7.10.2.3 Example

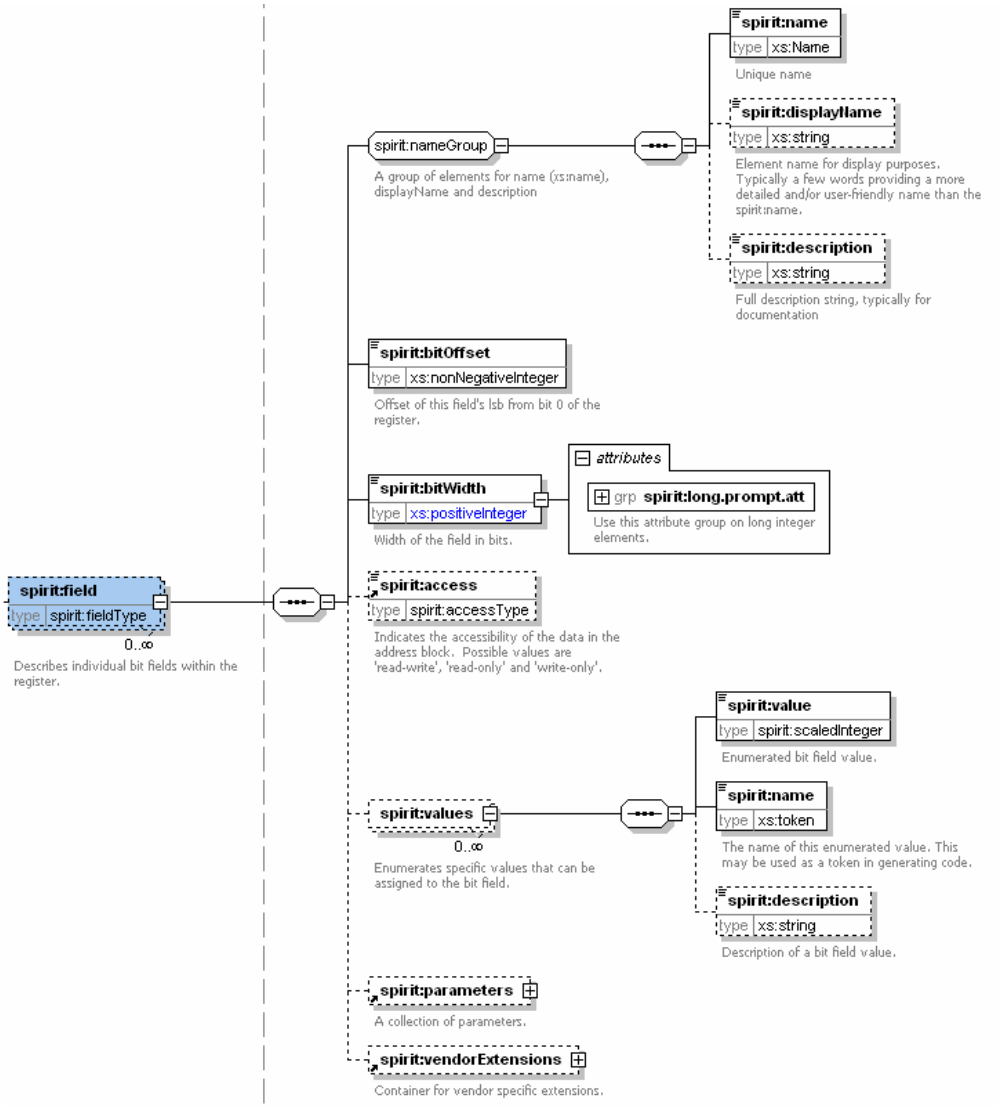
The following example shows a reset value. A register with this reset value will have its bottom eight bits set to `0` on reset.

```
<spirit:reset>
   <spirit:value>0</spirit:value>
   <spirit:mask>0xFF</spirit:mask>
</spirit:reset>
```

7.10.3 Register bit-fields

7.10.3.1 Schema

The following schema details the information contained in the **field** element, which may appear as an element inside the **register** element. This element describes a bit field of a register.



7.10.3.2 Description

A **field** element of a **register** describes a smaller bit-field of a register. This element contains the following elements.

a) ***nameGroup*** group includes the following. See X.Y.Z.

   1) **name** (mandatory) identifies the register.

   2) **displayName** (optional) allows a short descriptive text to be associated with the register.

   3) **description** (optional) allows a textual description of the register.

b) **bitOffset** (mandatory) describes the offset (from bit 0 of the register) where this bit-field starts. The **bitOffset** element is of type ***nonNegativeInteger***.

c) **bitWidth** (mandatory) is the width of the field, counting in bits. The bitWidth element is of type ***postiveInteger***. The **bitWidth** element is configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

d) **access** (optional) indicates the accessibility of the field: ***read-write***, ***read-only***, or ***write-only***. If this is not present, the access is inherited from the register.

e) **values** (optional) lists the set of legal values that may be written to the bit-field. This is an unbounded list, each containing 3 subelements.

   1) **value** (mandatory) is the value for the bit filed. The **value** element is of type ***scaledInteger***.

   2) **name** (mandatory) is a symbolic name for the value. The **name** element is of type ***token***.

   3) **description** (optional) is a textual description for the value. The **description** element is of type ***string***.

f) **parameters** (optional) details any additional parameters that describe the **field** for generator usage. See X.Y.Z.

g) **vendorExtensions** (optional) adds any extra vendor-specific data related to this field.

See also: SCR 7.2 and SCR 7.4.

7.10.3.3 Example

The following example shows a bit field with its sub-elements.

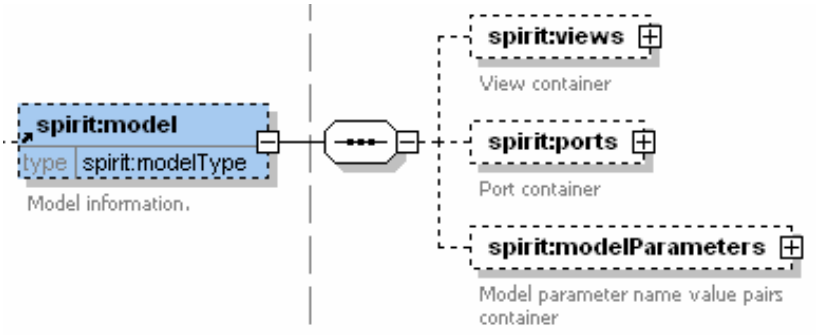
```
<spirit:field>
   <spirit:name>paritySelect</spirit:name>
   <spirit:displayName>Parity Select</spirit:displayName>
   <spirit:description>Selects parity polarity (0=odd parity, 1=even
   parity)</spirit:description>
   <spirit:bitOffset>4</spirit:bitOffset>
   <spirit:bitWidth>1</spirit:bitWidth>
   <spirit:access>read-write</spirit:access>
   <spirit:values>
      <spirit:value>0</spirit:value>
      <spirit:name>oddParity</spirit:name>
      <spirit:description>oddParity</spirit:description>
   </spirit:values>
   <spirit:values>
      <spirit:value>1</spirit:value>
      <spirit:name>evenParity</spirit:name>
      <spirit:description>evenParity</spirit:description>
   </spirit:values>
</spirit:field>
```

7.11 Models

7.11.1 Model

7.11.1.1 Schema

The following schema details the information contained in the **model** element, which may appear as an element inside the **component** or **abstractor** element.



7.11.1.2 Description

The **model** element describes the views, ports and model related parameters of a component or abstractor. A An object may A model element may contain the following.

a) **views** (optional) contains a list of all the views for this object. An object may have many different views. An RTL view may describe the source hardware module/entity with its pin interface; a SW view may define the source device driver C file with its `.h` interface; a documentation view may define the written specification of this IP. See 7.11.2.

b) **ports** (optional) contains the list of ports for this object. A ports is and external connection from the object. An object may only have one set of ports that must be valid for all view. See 7.11.3.

c) **modelParameters** (optional) contains a list of parameters that are needed to configure a model implementation specified in a view. An object may only have one set of model parameters that must be valid for all views. See 7.11.18.

7.11.1.3 Example

This shows a model section for a Timer component describing the view of the IP in terms of compatibility, language, file set reference, and model name.

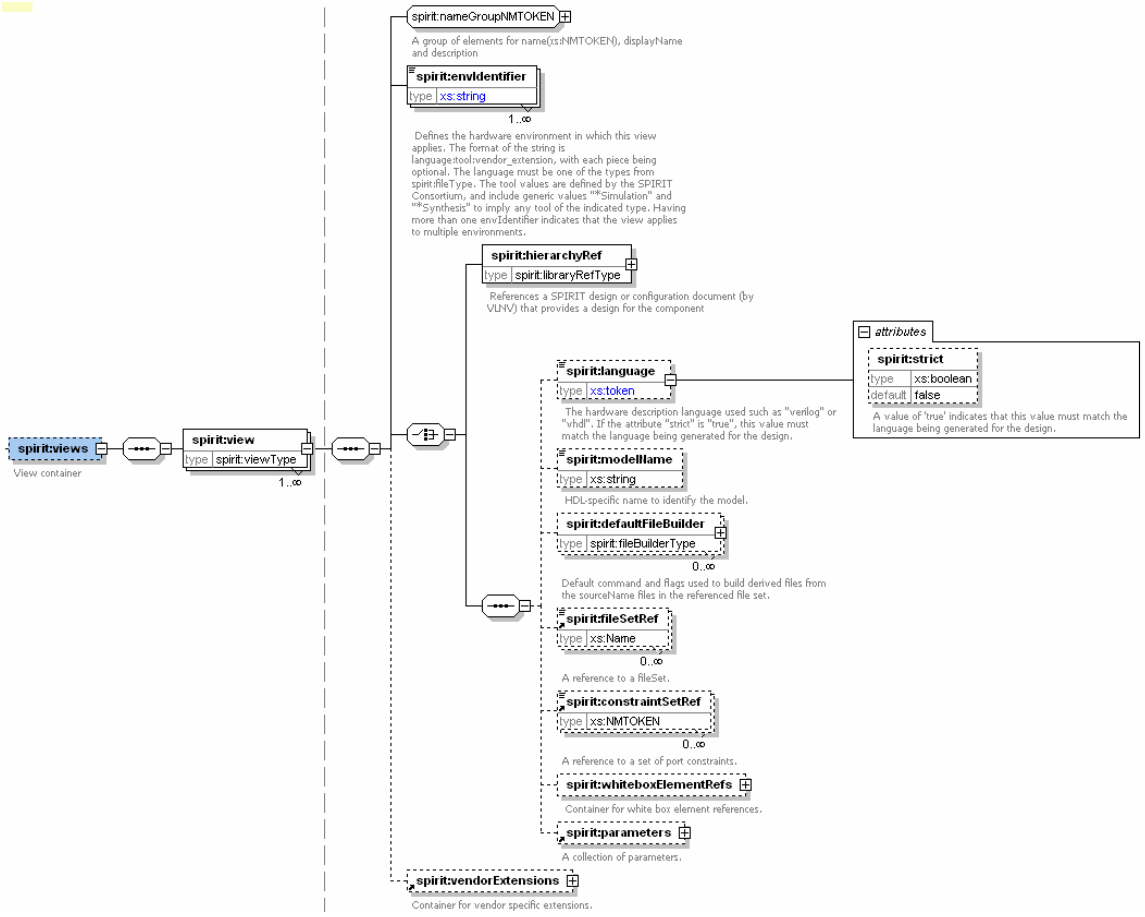
```
<spirit:model>
   <spirit: ports>
   ...
   </spirit: ports>
   <spirit:modelParameters>
   ...
   </spirit:modelParameters>
   <spirit:views>
      <spirit:view>
         <spirit:name>VHDL</spirit:name>
         <spirit:envIdentifier>:modelsim.mentor.com:</spirit:envIdentifier>
         <spirit:envIdentifier>:ncsim.cadence.com:</spirit:envIdentifier>
         <spirit:language spirit:strict="true">vhdl</spirit:language>
```

```
            <spirit:fileSetRef>fs-vhdlSource</spirit:fileSetRef>
            <spirit:modelName>leon2_Timers(struct)</spirit:modelName>
        <spirit:view>
    </spirit:views>
</spirit:model>
```

7.11.2 Views

7.11.2.1 Schema

The following schema details the information contained in the **views** element, which may appear as an element inside a **model** element. This element may contain one or more **view** elements.



7.11.2.2 Description

A **views** element describes an unbounded set of **view** elements. Each **view** element specifies a representation level of a component. It contains the following elements.

a) ***nameGroupNMToken*** group includes the following. See X.Y.Z.

   1) **name** (mandatory) identifies the view. The **name** element is of type ***NMTOKEN***.

   2) **displayName** (optional) allows a short descriptive text to be associated with the view. The **displayName** element is of type ***string***.

3) **description** (optional) allows a textual description of the view. The **description** element is of type ***string***.

b) **envIdentifier** designates and qualifies information about how this model view is deployed in a particular tool environment. The format of the element is a string with three fields separated by two colons [**:**] in the format of *Language*:*Tool*:*VendorSpecific*. The regular expression which is used to check the string is [A-Za-z0-9_+*\.]*:[A-Za-z0-9_+*\.]*:[A-Za-z0-9_+*\.]* The sections are:

i) *Language* indicates this view may be compatible with a particular tool, but only if that language is supported in that tool, e.g., different versions of some simulators may support two or more languages. In some cases, knowing the tool compatibility is not enough and may be further qualified by language compatibility, e.g., a compiled HDL model may work in a VHDL-enabled version of a simulator, but not in a SystemC-enabled version of the same simulator.

ii) *Tool* indicates this view contains information that is suitable for the named tool. This might be used if this view references data that is tool-specific and would not work generically, e.g., HDL models that use simulator-specific extensions.

Vendors shall publish lists of approved tool identification strings. These strings shall contain the tool name, as well as the company's domain name, separated by dots. Some examples of well-formed tool entries are:

```
designcompiler.synopsys.com
ncsim.cadence.com
modelsim.mentor.com
```

This field can alternatively indicate generic tool family compatibility, such as `*Simulation` or `*Synthesis`. To support transportability of created data files, it is important to use the published, generally recognized, tool designation when referencing a tool. See also www.The SPIRITconsortium.org.

iii) *VendorSpecific* can be used to further qualify tool and language compatibility. This can be used to indicate additional processing information may be required to use this model in a particular environment. For instance, if the model is a SWIFT simulation model, the appropriate simulator interface may need to be enabled and activated.

Any or all of the **envIdentifier** fields may be used. Where there are multiple environments for which a particular **view** is applicable, multiple **envIdentifier** elements can be listed.

c) The implementation details for this view can have two possibities. The first is a hierarchical view which uses the hierarchyRef element.

1) **hierarchyRef** (mandatory) references a hierarchical design from a view of a component. This element is required only if the view is used to reference a hierarchical design. The **hierarchyRef** element is of type ***libraryRefType*** (see X.Y.Z), it contains four attributes to specify a unique VLNV. See 6.2 on bus definitions.

i) **vendor** attribute (mandatory) identifies the owner of referenced description.

ii) **library** attribute (mandatory) identifies a library of referenced description.

iii) **name** attribute (mandatory) identifies a name of referenced description.

iv) **version** attribute (mandatory) identifies a version of referenced description.***addressSpecifier*** group includes the following.

d) The second possibility of a view is to reference a file set.

1) **language** (optional) specifies the hardware description language used for a specific view, for example, `verilog` or `vhdl`. The **language** element is of type ***token***. This may have an attribute **strict** (optional) of type Boolean; if **true** the language shall be strictly enforced. The default is **false**.

2) **modelName** (optional) is a language-specific identifier of the model. In VHDL's case, this may hold the top-level entity name and the architecture name or the configuration name;

whereas, in Verilog's case, this may simply hold the name of the module. The **modelName** element is of type ***string***.

3) **defaultFileBuilder** (optional) is an unbounded list of default file builder options for the **fileSets** referenced in this **view**. See 7.13.1.

4) **fileSetRef** (optional) is an unbounded list of references to **fileSets** used by this **view**.

5) **constraintSetRef** (optional) is an unbounded list of references to constraint sets, valid timing constraints for a view. **constraintsSets** are defined for **wire** style **ports**.

6) **whiteboxElementRefs** (optional) contains references to whitebox elements of a component that are vild for this view. If the view contains an implementation of any of the whitebox elements for the component, the **view** section shall include a reference to that **whitebox** element, with a string providing a language-dependent path to enable the DE to access the **whitebox** element. See 7.15.

7) **parameters** (optional) details any additional parameters that describe the **view** for generator usage. See X.Y.Z.

e) **vendorExtensions** (optional) adds any extra vendor-specific data related to the view.

See also: SCR 12.3.

7.11.2.3 Example

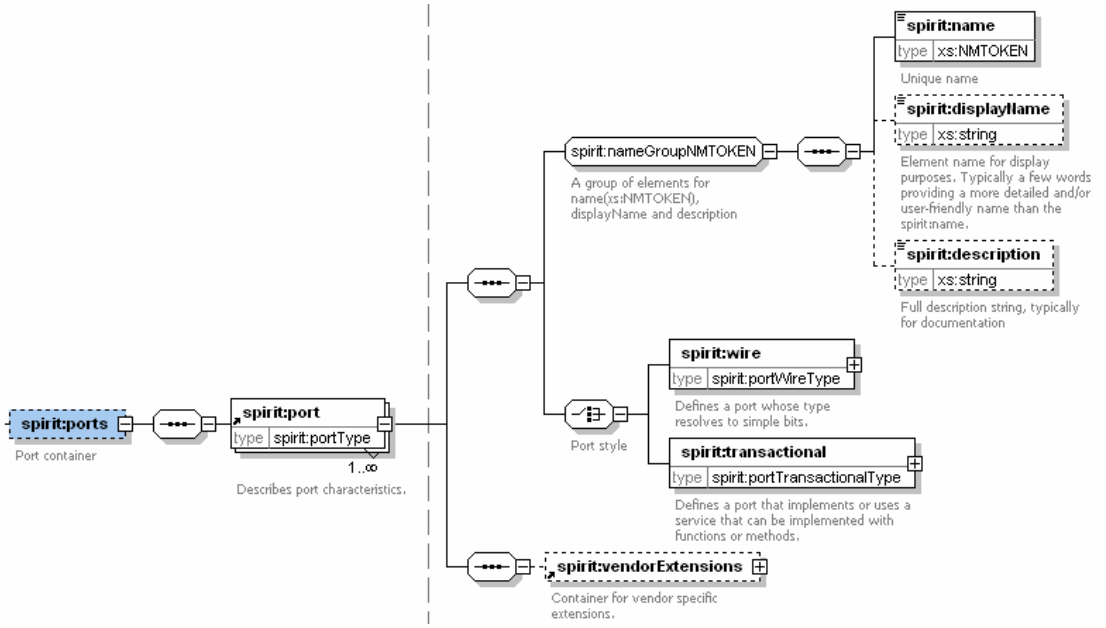
The following is an example of the **view** element with the name of `vhdlsource`.

```
<spirit:views>
   <spirit:view>
      <spirit:name>vhdlsource</spirit:name>
      <spirit:envIdentifier>:modelsim.mentor.com:</spirit:envIdentifier>
      <spirit:envIdentifier>:ncsim.cadence.com:</spirit:envIdentifier>
      <spirit:envIdentifier>:vcs.synopsys.com:</spirit:envIdentifier>
      <spirit:envIdentifier>:designcompiler.synopsys.com:
         </spirit:envIdentifier>
      <spirit:language>vhdl</spirit:language>
      <spirit:modelName>leon2_Uart(struct)</spirit:modelName>
      <spirit:fileSetRef>fs-vhdlSource</spirit:fileSetRef>
      <spirit:constraintSetRef>normal</spirit:constraintSetRef>
   </spirit:view>
</spirit:views>
```

7.11.3 Component ports

7.11.3.1 Schema

The following schema defines the information contained in the **ports** element, which may appear within a **component**.



7.11.3.2 Description

The **ports** element defines an unbounded list of **port** elements. Each port element describe a single external port on the ocmponent or abstractor.

a) ***nameGroupNMToken*** group includes the following. See X.Y.Z.

   1) **name** (mandatory) identifies the port. Each **port** shall be uniquely identified. The **name** element is of type ***NMTOKEN***.

   2) **displayName** (optional) allows a short descriptive text to be associated with the port. The **displayName** element is of type ***string***.

   3) **description** (optional) allows a textual description of the port. The **description** element is of type ***string***.

b) Each **port** shall be described as a **wire** or **transactional** port.

   1) **wire** (mandatory) defines ports that transport purely binary values or vectors of binary values. See 7.11.4.

   2) **transactional** (mandatory) defines all other style ports, typically used for transactionl level modeling (TLM). See 7.11.16.1.

c) **vendorExtensions** (optional) adds any extra vendor-specific data related to the port.

7.11.3.3 Example

This example shows a component with a wire port (`clk`) and two transactional ports (`initiator` and `target`).

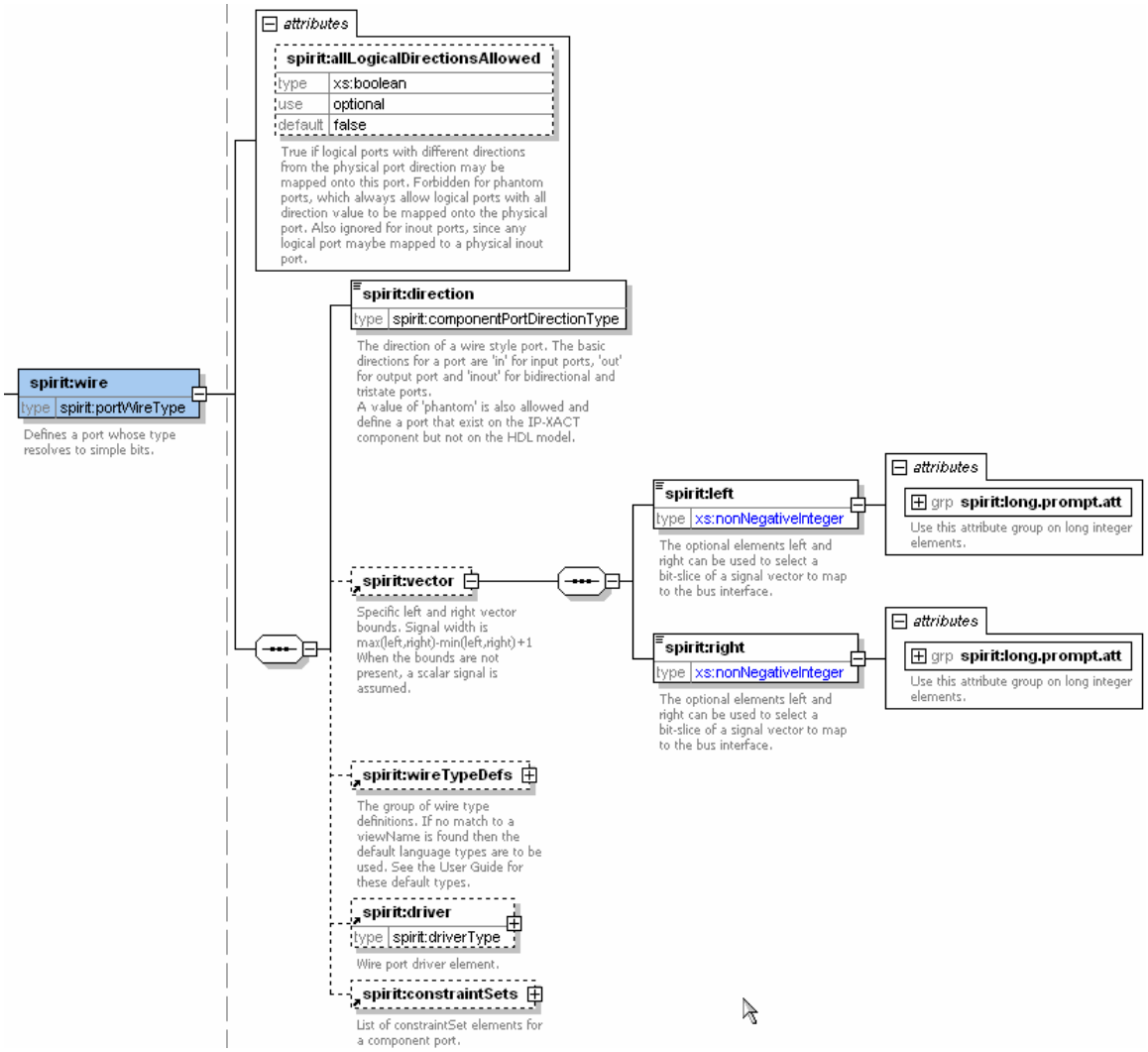
```
<spirit:ports>
   <spirit:port>
      <spirit:name>clk</spirit:name>
      <spirit:wire>
         <spirit:direction>in</spirit:direction>
      </spirit:wire>
   </spirit:port>
```

```
        <spirit:port>
            <spirit:transactional>
                <spirit:name>initiator</spirit:name>
                <spirit:service>
                    <spirit:initiative>requires</spirit:initiative>
                    <spirit:serviceTypeDefs>
                        <spirit:serviceTypeDef>
                            <spirit:typeName>read_write_if</spirit:typeName>
                        </spirit:serviceTypeDef>
                    /spirit:serviceTypeDefs>
                </spirit:service>
            </spirit:transactional>
        </spirit:port>
        <spirit:port>
            <spirit:transactional>
                spirit:name>target</spirit:name>
                <spirit:service>
                    <spirit:initiative>provides</spirit:initiative>
                        <spirit:serviceTypeDefs>
                            <spirit:serviceTypeDef>
                                <spirit:typeName>read_write_if
                            </spirit:typeName>
                        </spirit:serviceTypeDef>
                    </spirit:serviceTypeDefs>
                </spirit:service>
            </spirit:transactional>
        </spirit:port>
    </spirit:ports>
```

7.11.4 Component wire ports

7.11.4.1 Schema

The following schema details the information contained in the **wire** element, which may appear as an element inside the top-level **component**/**model**/**port** element.



7.11.4.2 Description

The **wire** element describes the properties for ports that are of a wire style. A port can come in two different styles, wire or transactional. A wire port applies for all scalar types (e.g., VHDL `std_logic` and Verilog `wire`) and vectors of scalars. A wire port transports purely binary values or vectors of binary values.

— Scalar types in VHDL also include integer and enumeration values. Scalars in IP-XACT only include binary values that relate to a single wire in an HW implementation.

— Since wire ports allow only binary values, IP-XACT does not support tri-state or multiple strength values.

The **wire** element contains the following elements.

a) **allLogicalDirectionsAllowed** (optional) attribute defines the possible legal combinations for the direction of ports between the component and the abstraction definition. See 6.2.

Table 4 shows the possible legal mappings from a component or abstractor port to the abstraction definition port through the bus interface port mappings when the attribute **allLogicalDirectionsAllowed** equal **false** is set.

When the attribute is instead set to **true**, all mapping values are possible (legal).

Table 4—allLogicalDirectionsAllowed="false"

Direction		logical direction		
		in	**out**	**inout**
Physical direction	**in**	legal	-	legal
	out	-	legal	legal
	inout	-	-	legal
	phantom	legal	legal	legal

a) **direction** (mandatory) specifies the direction of this port: **in** for input ports, **out** for output ports, and **inout** for bidirectional and tri-state ports. **phantom** can also be used to define a port which only exists on the IP-XACT component, but not on the implementation referenced from the view.

b) **vector** (optional) determines if this port is a scalar port or a vectored port. The **left** and **right** vector bounds elements inside the **vector** element are those specified in the implementation source. The port width is max(left,right) - min(left,right) +1. The **left** and **right** elements are of type ***nonNegativeInteger***. The **left** and **right** elements are configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

The **left** element means first boundary, the **right** element, the second boundary. **left** may be larger than **right** and that **left** may be the MSB or LSB (**right** being the opposite). The **left** and **right** elements are the (bit) rank of the left-most and right-most bits of the port.

When the **vector** element is present and the **left** and **right** elements are not equal, the port is defined as a *multi-bit vector port*. When the **vector** element is present and the **left** and **right** elements are equal, the port is defined as a *single-bit vector port*. When the **vector** element and the **left** and **right** elements are not present, the port is defined as a *scalar port*.

c) **wireTypeDefs** (optional) describes the ports type as defined by the implementation, see 7.11.5.

d) **driver** (optional) defines a driver which may be attached to this port if no other object is connected to this port. This allows the IP to define the default state of unconnected inputs. A wire style port may only define a **driver** element for a port if the direction of the port is **in** or **inout**. See also 7.11.6

e) **constraintSets** (optional) defines multiple set of constraints on a port used for synthesis or other operations. See 7.11.11.

7.11.4.3 Example

The following examples show how the vector elements are used when mapping to an HDL language.

```
reset: in std_logic; -- VHDL
```

would be defined with no **left** or **right** elements under the **vector** element.

```
<spirit:wire>
   <spirit:direction>in</spirit:direction>
</spirit:wire>
```

Whereas

```
data: out std_logic_vector(29 downto 3); -- VHDL
```

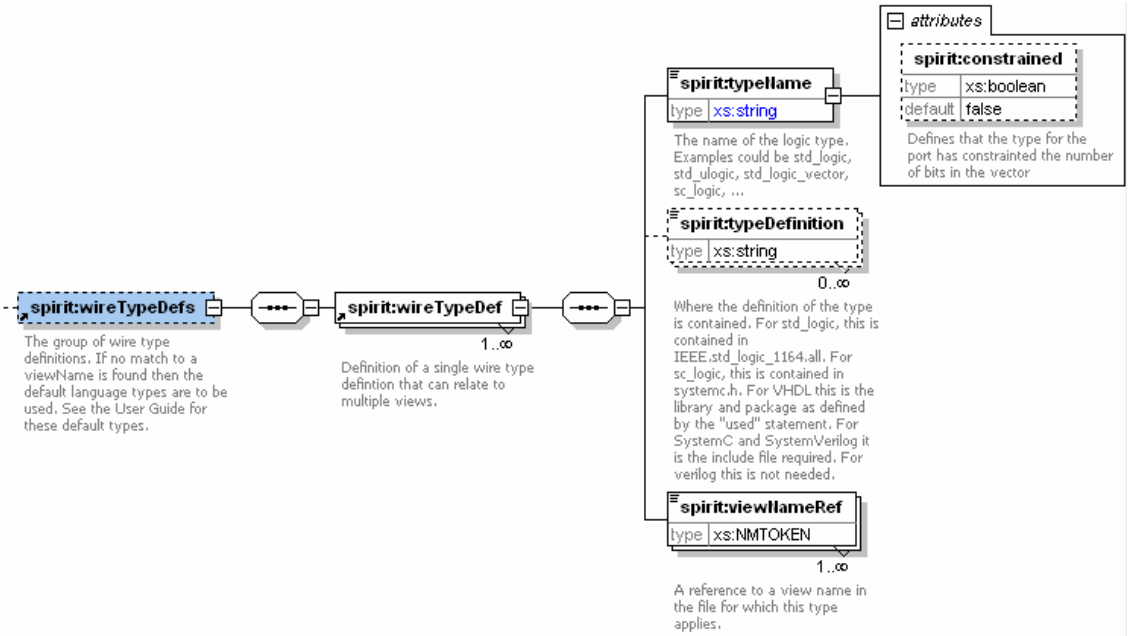
would be defined in IP-XACT as `left=29` and `right=3` with all bits in descending order.

```
<spirit:wire>
    <spirit:direction>out</spirit:direction>
    <spirit:vector>
        <spirit:left>29</spirit:left>
        <spirit:right>3</spirit:right>
    </spirit:vector>
</spirit:wire>
```

7.11.5 Component wireTypeDef

7.11.5.1 Schema

The following schema details the information contained in the **wireTypeDef** element, which may appear as an element inside the top-level wire style **port** element. These elements define the definition type name, where the type is defined, and which views of a component or an abstractor use this type.



7.11.5.2 Description

The **wireTypeDefs** element describes the type properties for a port per view of a component or abstractor. There can be an unbounded series of **wireTypeDef**s defined for each port, allowing the type properties to be defined differently for each view. **wireTypeDef** contains the following elements.

a) **typeName** (mandatory) defines the name of the type for the port. For VHDL, some typical values would be `std_logic` and `std_ulogic`.

   1) **constrained** (optional) attribute indicates the type of definition that is used for the array port. The **constrained** element is of type Boolean. When set to **true**, this indicates that the port of the type is constrained and the indices are not needed when the type is used. The default is **false**, which indicates that the definition has not constrained the number of bits. See 7.11.5.2.1 and 7.11.5.2.2.

b) **typeDefinition** (optional) is defined by IP-XACT per language. Table 5 shows some examples. There can be multiple **typeDefinition**s for each port. The **typeDefinition** element is of type ***string***..

Table 5—typeDefinition examples

Language	Meaning
VHDL	"Use" statement text (`IEEE.std_logic_1164.all`).
Verilog	Nothing needed, no meaning.
SystemC	Include file name (`systemc.h`).
SystemVerilog	Include file name (if the name does not contain a `:`); import package name (if the name contains a `:`).

c) **viewNameRef** (mandatory) maps the correct type properties to the correct view. Multiple views can use the same set of type properties by specifying multiple **viewNameRef** elements. The **viewNameRef** must match a **view**/**name** in the containing object. The **viewNameRef** element is of type ***NMTOKEN***.

7.11.5.2.1 Constrained array type

A *constrained array type* is a type for which the indices of the array have been specified in the definition.

```
type BYTE is array (7 downto 0) of std_logic;
entity example is
  port(
    A: out BYTE;
    B: in BYTE
  );
end example;
```

Also, the definition of `port A` in an IP-XACT file contains the indices in XML to designate the width so these types below can be mixed in the same component.

```
<spirit:port>
   <spirit:name>A</spirit:name>
   <spirit:wire>
      <spirit:vector>
         <spirit:left>7</spirit:left>
         <spirit:right>0</spirit:right>
      </spirit:vector>
      <spirit:typeDefs>
         <spirit:typeDef>
            <spirit:typeName spirit:constrained="true">BYTE
               </spirit:typeName>
            <spirit:typeDefinition>MYLIB.MYPKG.all</spirit:typeDefinition>
            <spirit:viewNameRef>VHDLsimView</spirit:viewNameRef>
         </spirit:typeDef>
      </spirit:typeDefs>
   </spirit:wire>
<spirit:port>
```

7.11.5.2.2 Unconstrained array type

An *unconstrained array type* is a type for which the indices of the array have not been specified in the definition, e.g.,

```
type std_logic_vector is array ( NATURAL RANGE <> ) of std_logic;

entity example is
  port(
    A: out std_logic_vector (7 downto 0);
    B: in std_logic_vector (7 downto 0)
  );
end example;
```

could be described in IP-XACT as

```
<spirit:port>
   <spirit:name>A</spirit:name>
   <spirit:wire>
      <spirit:vector>
         <spirit:left>7</spirit:left>
         <spirit:right>0</spirit:right>
      </spirit:vector>
      <spirit:typeDefs>
         <spirit:typeDef>
            <spirit:typeName spirit:constrained="false">BYTE
               </spirit:typeName>
            <spirit:typeDefinition>MYLIB.MYPKG.all</spirit:typeDefinition>
            <spirit:viewNameRef>VHDLsimView</spirit:viewNameRef>
         </spirit:typeDef>
   </spirit:wire>
<spirit:port>
```

7.11.5.2.3 Defaults

wireTypeDefs do not need to be defined for every view of a port. IP-XACT provides for these defaults based on the language of the view, as shown in Table 6. For those languages not shown here, no defaults can be presumed.

Table 6—View defaults

Language	Single bit	Vectors
VHDL	`std_logic`	`std_logic_vector`
Verilog	`wire`	`wire`
SystemC	`sc_logic`	`sc_lv`
SystenVerilog	`logic`	`logic`

7.11.5.2.4 Rules

— A view name may only appear once in all the ports **viewNameRef** elements.
— If the view name is not found in a **viewNameRef**, the default type properties apply (seeTable 6).

7.11.5.3 Example

See the examples in 7.11.5.2.2.

7.11.6 Component driver

7.11.6.1 Schema

The following schema details the information contained in the **driver** element, which may appear as an element inside the top-level wire style **port** element. This element defines the type and value(s) to drive on this port when it is not connected in a design.

7.11.6.2 Description

The **driver** element shall contain one of three different types of drivers that can be applied to a wire port of a component or abstractor.

a) **defaultValue** (optional) specifies a static logic value for this port. The **defaultValue** can specify a simple 1-bit wire port or a vectored wire port. The **defaultValue** element is of type ***scaledNonNegativeInteger***. The **defaultValue** element is configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

b) **clockDriver** (optional) specifies a repeating high-low waveform of this port. See 7.11.7.

c) **singleShotDriver** (optional) specifies a non-repeating high-low waveform for this port. See 7.11.8.

A driver element shall not be defined for a **wire** style port with a **direction** element of **out**.

7.11.6.3 Example

This example shows a default value of `0x0F` set for a vectored wire port named `scaler`.

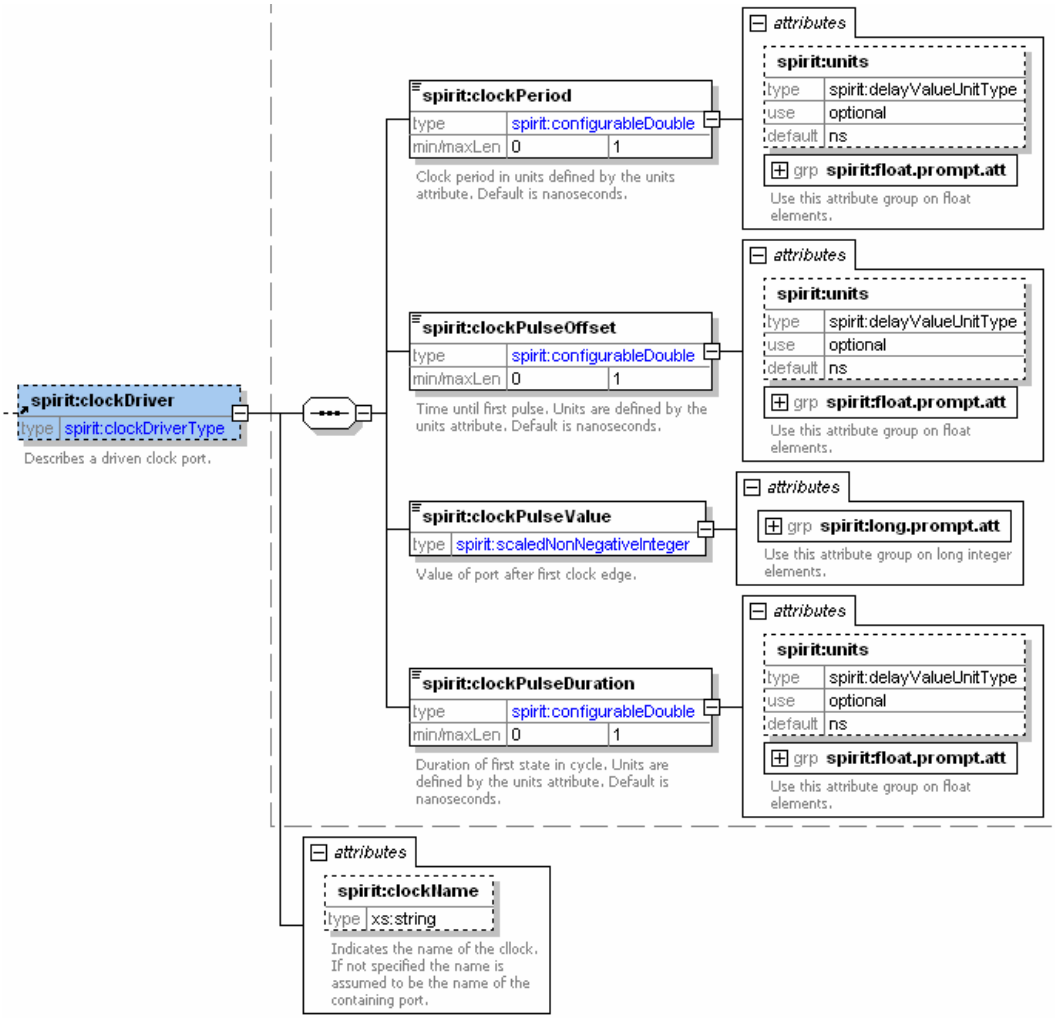
```
<spirit:port>
   <spirit:name>scaler</spirit:name>
   <spirit:wire>
      <spirit:direction>in</spirit:direction>
      <spirit:vector>
         <spirit:left>7</spirit:left>
         <spirit:right>0</spirit:right>
      </spirit:vector>
      <spirit:driver>
         <spirit:defaultValue>0x0F</spirit:defaultValue>
      </spirit:driver>
   </spirit:wire>
```

```
</spirit:port>
```

7.11.7 Component driver/clockDriver

7.11.7.1 Schema

The following schema details the information contained in the **clockDriver** element, which may appear as an element inside the top-level wire style **port**/**driver** element. This element defines the properties of a clock waveform or repeating high-low waveform.



7.11.7.2 Description

The **clockDriver** element contains four elements that describe the properties of a clock waveform. These are also depicted in Figure 10.

a) **clockPeriod** (mandatory) specifies the overall length (in time) of one cycle of the waveform. The **clockPeriod** element is of type ***configurableDouble***. The **clockPeriod** element is configurable with attributes from ***float.prompt.att***, see X.Y.Z on configuration. This element also contains a **units** (optional) attribute for specifying the units of their time values: **ns** (the default) and **ps**. **ns** stands for nanosecond and is equal to 10^{-9} seconds. **ps** stands for picosecond and is equal to 10^{-12} seconds.

b) **clockPulseOffset** (mandatory) specifies the time delay from the start of the waveform to the first transition. The **clockPulseOffset** element is of type ***configurableDouble***. The **clockPulseOffset** element is configurable with attributes from ***float.prompt.att***, see X.Y.Z on configuration. This element also contains a **units** (optional) attribute for specifying the units of their time values: **ns** (the default) and **ps**. **ns** stands for nanosecond and is equal to 10^{-9} seconds. **ps** stands for picosecond and is equal to 10^{-12} seconds.

c) **clockPulseValue** (mandatory) specifies the logic value to which the signal transitions. This value is also the opposite of the value from which the waveform will start. The **clockPulseValue** element is of type ***scaledNonNegativeInteger***. The **clockPulseValue** element is configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

d) **clockPulseDuration** (mandatory) specifies how long the waveform remains at the value specified by **clockPulseValue**. The **clockPulseDuration** element is of type ***configurableDouble***. The **clockPulseDuration** element is configurable with attributes from ***float.prompt.att***, see X.Y.Z on configuration. This element also contains a **units** (optional) attribute for specifying the units of their time values: **ns** (the default) and **ps**. **ns** stands for nanosecond and is equal to 10^{-9} seconds. **ps** stands for picosecond and is equal to 10^{-12} seconds.

e) **clockName** (optional) attribute specifies a name for the clock driver. If this is not defined, the name of the port to which this **clockDriver** is applied shall be used.

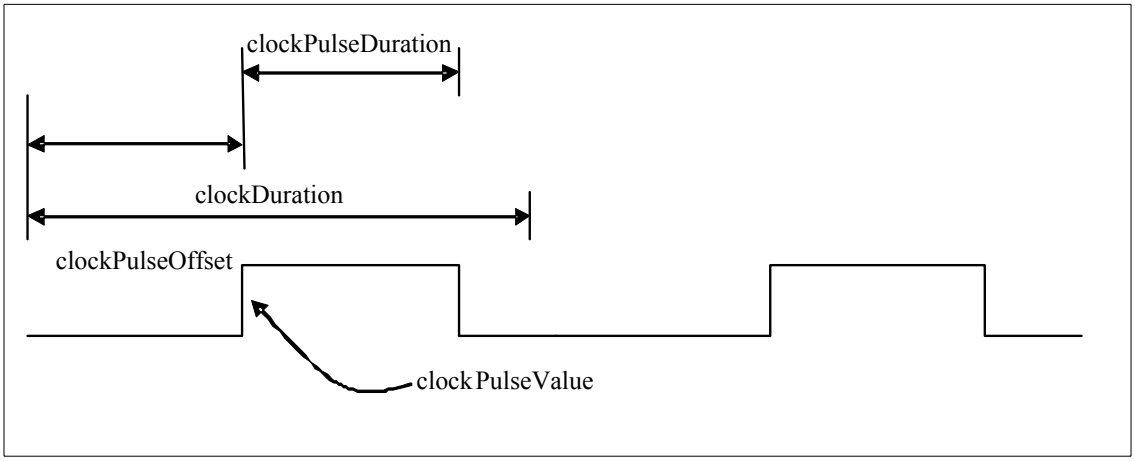


Figure 10—clockDriver elements

7.11.7.3 Example

This is an example of a clock driver set on the wire port named `clk`. The clock starts off in the `logic 0` state for `4 ns`, then transitions to the `logic 1` state for `4 ns`. This cycle is the repeated forever.

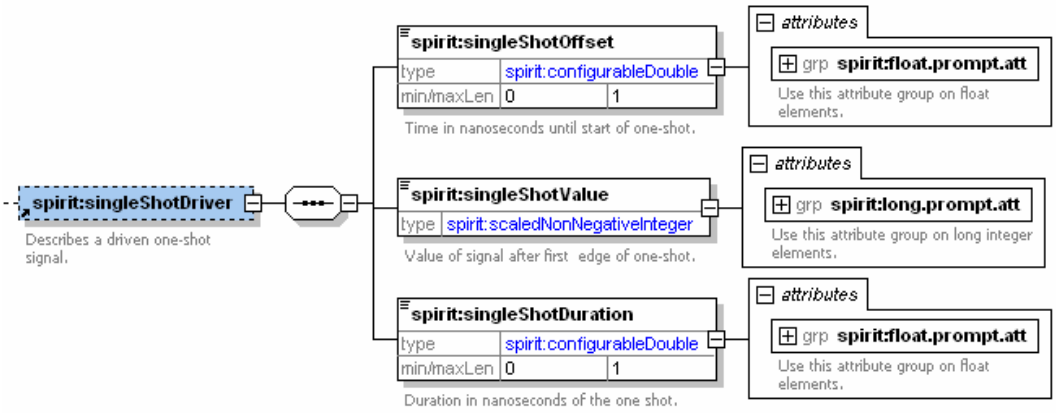
```
<spirit:port>
   <spirit:name>clk</spirit:name>
   <spirit:wire>
      <spirit:direction>in</spirit:direction>
      <spirit:driver>
         <spirit:clockDriver spirit:clockName="clk">
            <spirit:clockPeriod>8</spirit:clockPeriod>
            <spirit:clockPulseOffset>4</spirit:clockPulseOffset>
            <spirit:clockPulseValue>1</spirit:clockPulseValue>
            <spirit:clockPulseDuration>4</spirit:clockPulseDuration>
         </spirit:clockDriver>
```

```
        </spirit:driver>
      </spirit:wire>
    </spirit:port>
```

7.11.8 Component driver/singleShotDriver

7.11.8.1 Schema

The following schema details the information contained in the **singleShotDriver** element, which may appear as an element inside the top-level wire style **port**/**driver** element. This element defines the properties of a single-shot waveform or non-repeating high-low waveform.



7.11.8.2 Description

The **singleShotDriver** element contains three elements that describe the properties of the waveform. These are also depicted in Figure 11.

a) **singleShotOffset** (mandatory) specifies the time delay from the start of the waveform to the transition. The **singleShotOffset** element is of type ***configurableDouble***. The **singleShotOffset** element is configurable with attributes from ***float.prompt.att***, see X.Y.Z on configuration. This element also contains a **units** (optional) attribute for specifying the units of their time values: **ns** (the default) and **ps**. **ns** stands for nanosecond and is equal to 10^{-9} seconds. **ps** stands for picosecond and is equal to 10^{-12} seconds.

b) **singleShotValue** (mandatory) specifies the logic value to which the signal transitions. This value is also the opposite of the value from which the waveform will start. This value is also the opposite of the value from which the waveform will start. The **singleShotValue** element is of type ***scaledNonNegativeInteger***. The **singleShotValue** element is configurable with attributes from ***long.prompt.att***, see X.Y.Z on configuration.

c) **singleShotDuration** (mandatory) specifies hog long the waveform remains at the value specified by **singleShotValue**. The **singleShotDuration** element is of type ***configurableDouble***. The **singleShotDuration** element is configurable with attributes from ***float.prompt.att***, see X.Y.Z on configuration. This element also contains a **units** (optional) attribute for specifying the units of their time values: **ns** (the default) and **ps**. **ns** stands for nanosecond and is equal to 10^{-9} seconds. **ps** stands for picosecond and is equal to 10^{-12} seconds.

d) These elements all have a group of attributes named **general.att** applied to them. These attributes are described in the general section of this document, they allow the user to change the values of these defaults in the design file for each instantiation of a component or change the design configu-

ration file for each instantiation of an abstractor. The two elements related to time (**singleShotDuration** and **singleShotOffset**) have a fixed time unit of nanoseconds (10^{-9} seconds).

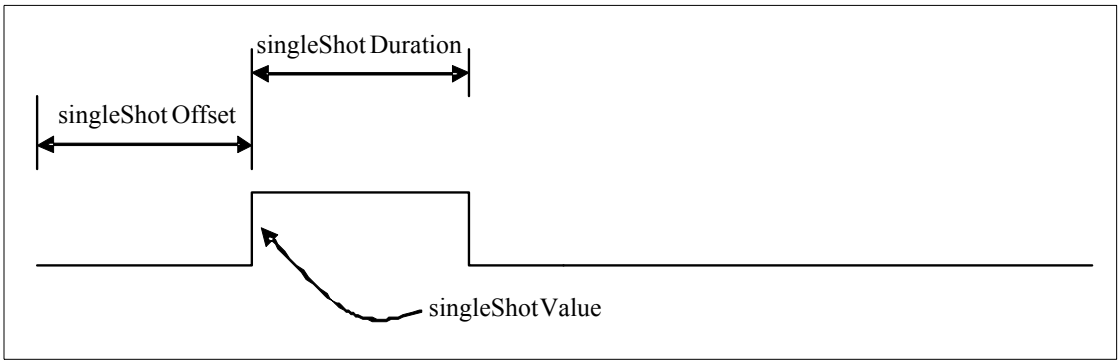


Figure 11—singleShotDriver elements

7.11.8.3 Example

This is an example of a single-shot driver set on the wire port named `reset`. The waveform starts off in the `logic 0` state for `100 ns` and then transitions to the `logic 1` state.

```
<spirit:port>
   <spirit:name>reset</spirit:name>
   <spirit:wire>
      <spirit:direction>in</spirit:direction>
      <spirit:driver>
         <spirit:singleShotDriver>
            <spirit:singleShotOffset>0</spirit:singleShotOffset>
            <spirit:singleShotValue>0</spirit:singleShotValue>
            <spirit:singleShotDuration>100</spirit:singleShotDuration>
         </spirit:singleShotDriver>
      </spirit:driver>
   </spirit:wire>
</spirit:port>
```

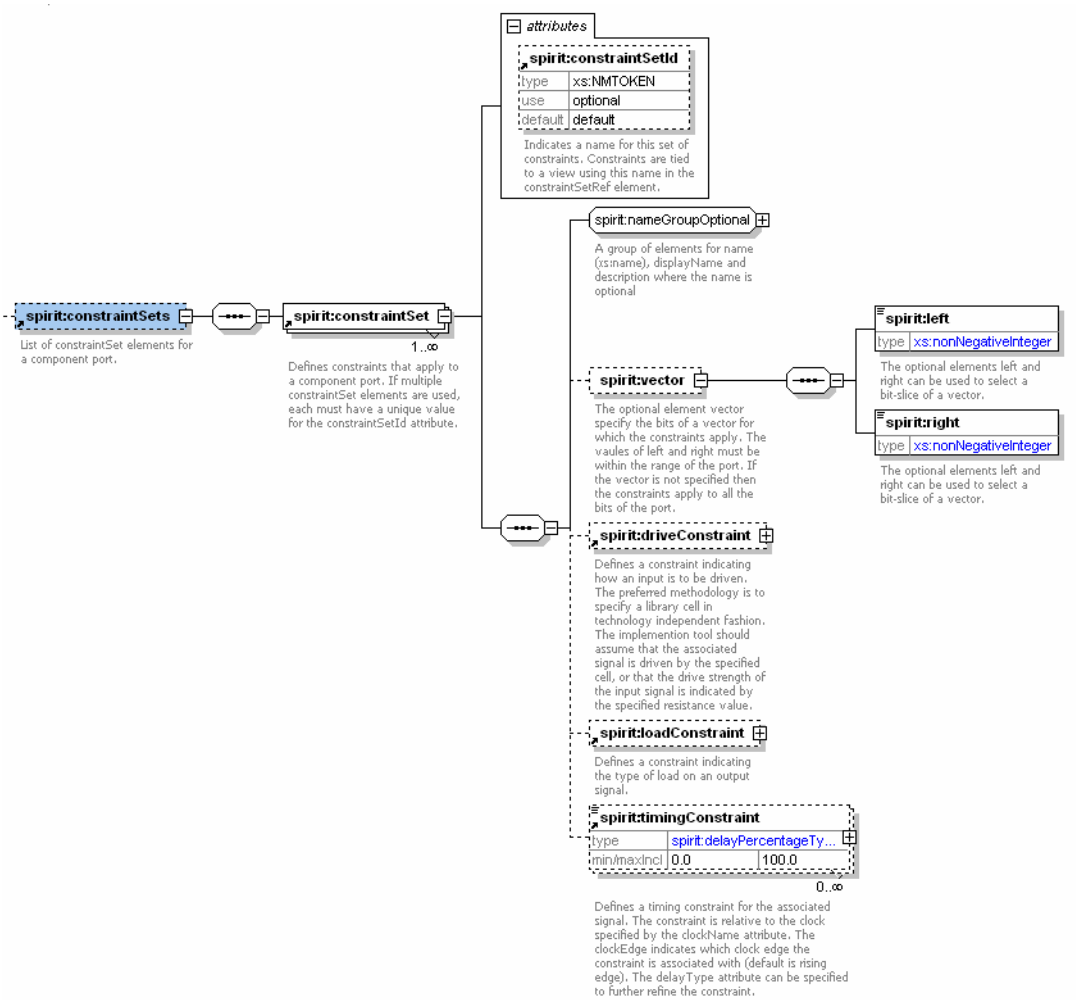
7.11.9 Implementation constraints

Implementation constraints can be defined to document requirements that need to be met by an implementation of the component. Constraints are defined in groups called *constraint sets* (in the IP-XACT element **port**/**wire**/**constraintSets**/**constraintSet**) so different constraints can be associated with different views of the component. A particular set of constraints is tied to a component view by the **constraintSetId** attribute in the constraint set and the matching **constraintSetRef** element in the view.

7.11.10 Component wire port constraints

7.11.10.1 Schema

The following schema defines the information contained in the **constraintSets** element, which may appear within a **wire** element within a component **port** element (**component**/**model**/**ports**/**port**/**wire**).



7.11.10.2 Description

The **constraintSets** element is used to define technology independent implementation constraints associated with the containing wire port of the component. The **constraintSets** element contains one or more **constraintSet** elements which define a set of constraints for the port. If more than one **constraintSet** element is present, each shall have a unique value for the **constraintSetId** attribute so each **constraintSet** can be uniquely referenced from a **view**. **constraintSet** also contains the following optional elements.

a) ***nameGroupOptional*** group includes the following. See X.Y.Z.
   1) **name** (optional) identifies the constraint set.
   2) **displayName** (optional) allows a short descriptive text to be associated with the constraint set.
   3) **description** (optional) allows a textual description of the constraint set.

b) **vector** (optional) determines if this port is a scalar port or a vectored port. The **left** and **right** vector bounds elements inside the **vector** element are those specified the bounds of the vector. The **left** and **right** elements are of type ***nonNegativeInteger***.

c) **driveConstraint** (optional) defines a driving constraint for this port. See 7.11.11 for details.

d) **loadConstraint** (optional) defines a load constraint for this port. See 7.11.12 for details.

e) **timingConstraint** (optional) defines a timing constraint relitive to a clock for this port. See 7.11.13 for details.

7.11.10.3 Example

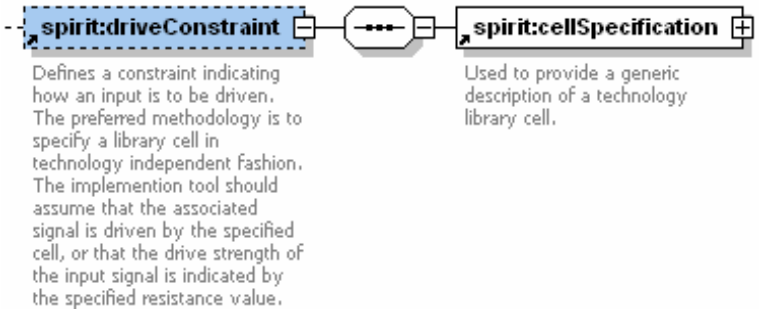
This example shows a port containing a single timing constraint appearing in two different constraint sets.

```
<spirit:port>
   <spirit:name>hgrant</spirit:name>
   <spirit:wire>
      <spirit:direction>in</spirit:direction>
      <spirit:constraintSets>
         <spirit:constraintSet spirit:constraintSetId="timing">
            <spirit:timingConstraint spirit:clockName="hclk">40
               </spirit:timingConstraint>
         </spirit:constraintSet>
         <spirit:constraintSet spirit:constraintSetId="area">
            <spirit:timingConstraint spirit:clockName="hclk">50
               </spirit:timingConstraint>
         </spirit:constraintSet>
      </spirit:constraintSets>
   </spirit:wire>
</spirit:port>
```

7.11.11 Port drive constraints

7.11.11.1 Schema

The following schema defines the information contained in the **driveConstraint** element, which may appear within a **modeConstraints** or **mirroredModeConstraints** element within a wire type port in an abstraction definition or within a **constraintSet** element within a wire type port in a component.



7.11.11.2 Description

The **driveConstraint** element defines a technology-independent drive constraint associated with the containing wire port of a component or the component port associated with the logical port within an abstraction definition if the **driveConstraint** element is contained within an abstraction definition. The actual constraint consists of a technology-independent specification of a library cell presumed to drive the input port. The **cellSpecification** element defines the cell (see 7.11.14).

The **driveConstraint** element is not valid on output port.

See also: SCR 14.1, SCR 14.3, and SCR 14.6.

7.11.11.3 Example

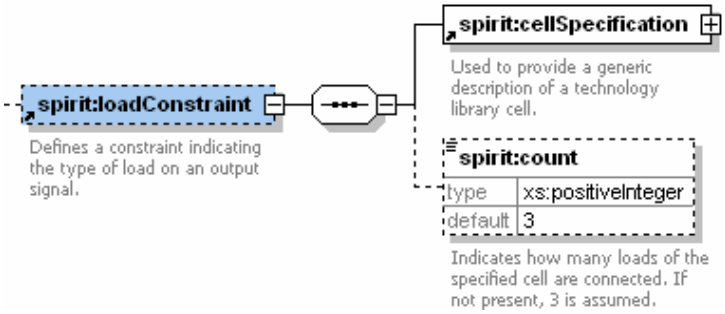
This example shows two different drive constraints. The first represents a median-strength D flop and the second a low-strength sequential cell.

```
<spirit:driveConstraint>
    <spirit:cellSpecification>
        <spirit:cellFunction>dff</spirit:cellFunction>
    </spirit:cellSpecification>
</spirit:driveConstraint>

<spirit:driveConstraint>
    <spirit:cellSpecification>
        <spirit:cellClass spirit:strength="low">sequential
        </spirit:cellClass>
    </spirit:cellSpecification>
</spirit:driveConstraint>
```

7.11.12 Port load constraints

7.11.12.1 Schema

The following schema element defines the information contained in the **loadConstraint** element, which may appear within a **modeConstraints** or **mirroredModeConstraints** element within a wire type port in an abstraction definition or within a **constraintSet** element within a wire type port in a component.



7.11.12.2 Description

The **loadConstraint** element defines a technology-independent load constraint associated with the containing wire port of a component or the component port associated with the logical port within an abstraction definition if the **loadConstraint** element is contained within an abstraction definition. The actual constraint consists of two parts, the technology-independent specification of a library cell and a count. **loadConstraint** also contains the following elements.

a) **cellSpecification** (mandatory) defines the library cell (see 7.11.14).

b) **count** (optional) indicates how many loads of the indicated type are modeled as if attached to the output port. The default is three loads. The **count** element is of type ***positiveInteger***.

The **loadConstraint** element is not valid on input ports.

See also: SCR 14.2, SCR 14.4, and SCR 14.5.

7.11.12.3 Example

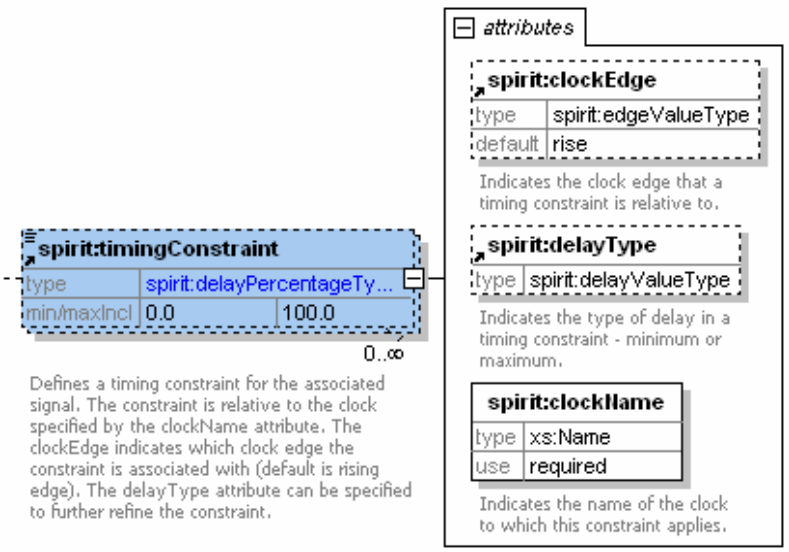
This example shows two different load constraints. The first is load consisting of three D flops of median strength and the second is a load consisting of four low-strength sequential cells.

```
<spirit:loadConstraint>
   <spirit:cellSpecification>
      <spirit:cellFunction>dff</spirit:cellFunction>
   </spirit:cellSpecification>
</spirit:loadConstraint>
<spirit:loadConstraint>
   <spirit:cellSpecification>
      <spirit:cellClass spirit:strength="low">sequential</spirit:cellClass>
   </spirit:cellSpecification>
   <spirit:count>4</spirit:count>
</spirit:loadConstraint>
```

7.11.13 Port timing constraints

7.11.13.1 Schema

The following schema defines the information contained in the **timingConstraint** element, which may appear within a **modeConstraints** or **mirroredModeConstraints** element within a wire type port in an abstraction definition or within a **constraintSet** element within a wire type port in a component.



7.11.13.2 Description

The **timingConstraint** element defines a technology-independent timing constraint associated with the containing wire port of a component or abstraction definition. Its of type ***delayPercentageType***, the value is a floating point number between `0` and `100` which represents the percentage of the cycle time to be allocated to the timing constraint on the port. If the component port is an input (or the port in an abstraction definition ends up mapping to a physical port with direction **in**), the timing constraint represents an input delay constraint; otherwise, it represents an output delay constraint. **timingConstraint** also contains the following attributes.

a) **clockEdge** (optional) specifies to which edge of the clock the constraint is relative. The default behavior is the constraint is relative to the rising edge of the clock. The **clockEdge** attribute may have two values **rise** (the default) or **fall**.

b) **delayType** (optional) restricts the constraint to applying to only best-case (minimum) or worst-case (maximum) timing analysis. By default, the constraint is applied to both. The **delayType** attribute may have two values **min** or **max**.

c) **clockName** (mandatory) specifies the delay constraint relative to the clock. The cycle time of the referenced clock is what actually determines the actual magnitude of the delay constraint (<clock cycle time> * 100 / <timing constraint element value>). The **clockName** element is of type ***Name***.

See also: SCR 14.7 and SCR 14.9.

7.11.13.3 Example

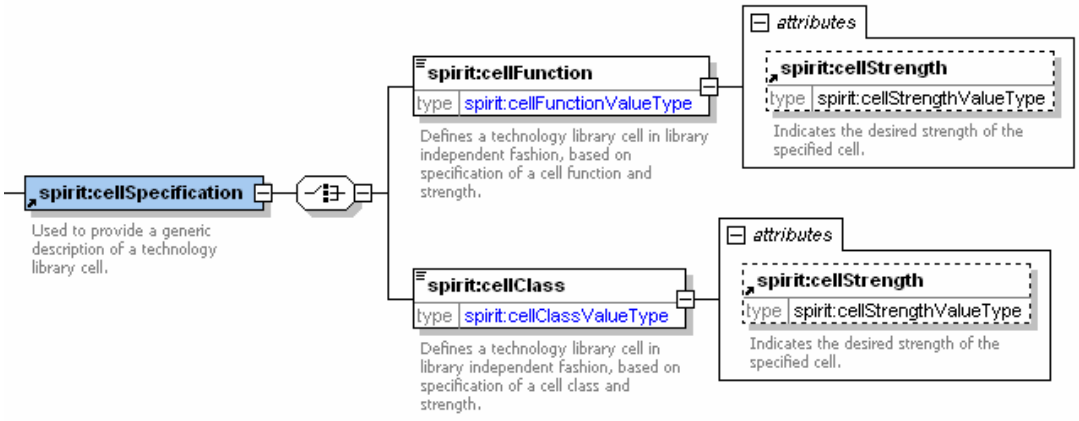
This example shows three basic timing constraints. The first indicates a delay of 40% of the clock `hclk`, relative to the rising edge of `hclk`, and applicable to both best and worst case timing analysis. The second indicates a delay of 30% of the clock `hclk`, relative to the falling edge of `hclk`, and applicable to best case timing. The third indicates a delay of 50% of the clock `hclk`, relative to the falling edge of `hclk`, and applicable to worst case timing.

```
<spirit:timingConstraint
    spirit:clockName="hclk">40</spirit:timingConstraint>
<spirit:timingConstraint spirit:clockName="hclk"spirit:clockEdge="fall"
    spirit:delayType="min">30</spirit:timingConstraint>
<spirit:timingConstraint spirit:clockName="hclk" spirit:clockEdge="fall"
    spirit:delayType="max">50</spirit:timingConstraint>
```

7.11.14 Load and drive constraint cell specification

7.11.14.1 Schema

The following schema defines the information contained in the **cellSpecification** element, which may appear within a **loadConstraint** or **driveConstraint** element indicating the type of cell to use in the constraint.



7.11.14.2 Description

The **cellSpecification** element defines a cell in a technology-independent fashion such that drive and load constraints can be defined without referencing a specific technology library. The cell is defined so a design environment can map it to an appropriate cell in a specific library when the actual constraint is generated. The **cellSpecification** element ahsll contain *one* of the following two elements.

a) **cellFunction** (mandatory) specifies a cell function from the user defined library. The **cellFunction** element shall be one of the following values: **nd2**, **buf**, **inv**, **mux21**, **dff**, **latch** or **xor2**. The **cellFunction** element contains a **cellStrength** (optional) attribute that provides the cell strength specification. The value shall be one of **low**, **median** (the default) or **high**. **median** implies the middle cell of all the cells that match the desired function, sorted by drive or load strength (as appropriate for the given constraint), is used.

b) **cellClass** (mandatory) specifies a cell class from the user defined library. The **cellClass** element shall be one of the following values: **combinational** or **sequential**. The **cellClass** element contains a **cellStrength** (optional) attribute that provides the cell strength specification. The value shall be one of **low**, **median** (the default) or **high**. **median** implies the middle cell of all the cells that match the desired class, sorted by drive or load strength (as appropriate for the given constraint), is used.

7.11.14.3 Example

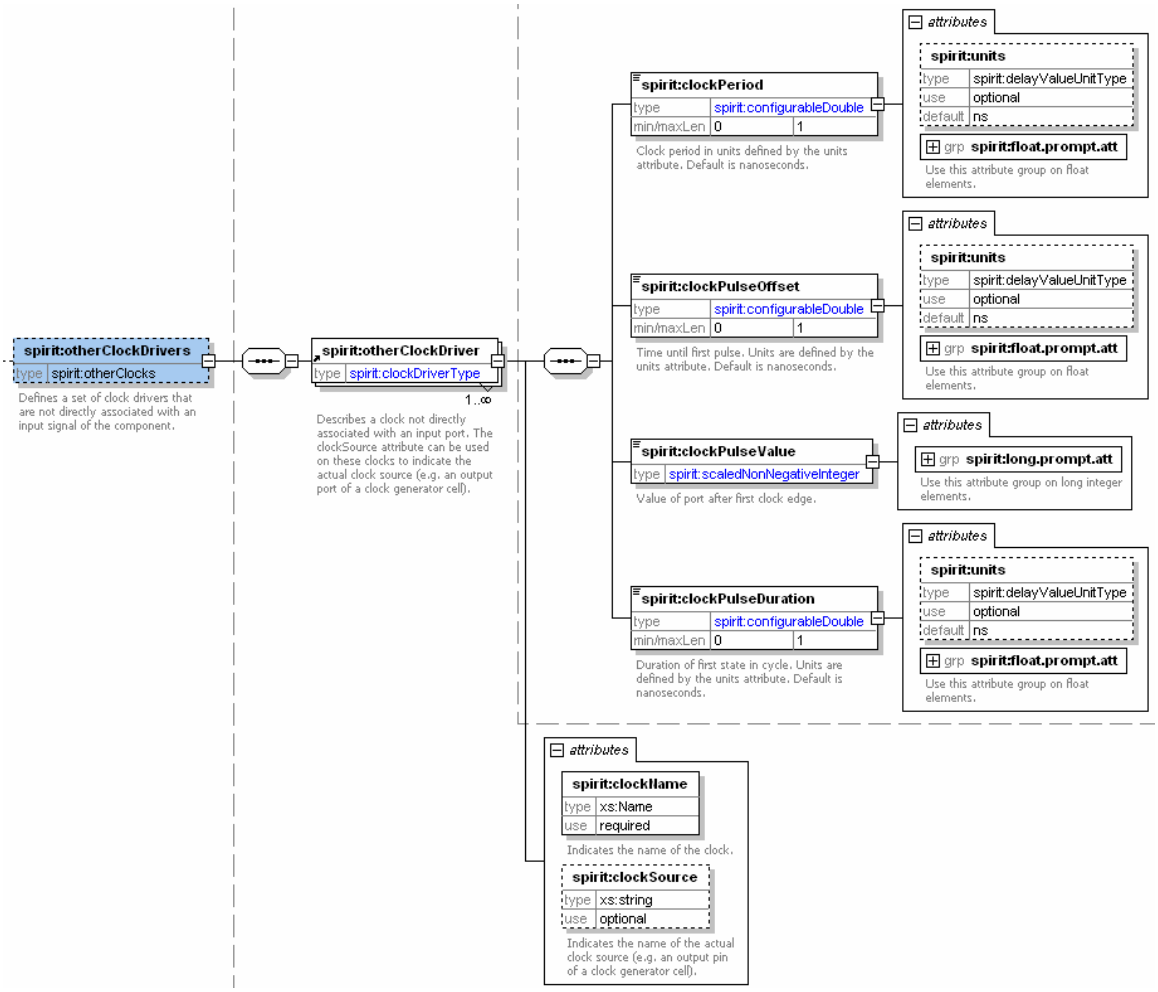
This example shows two different variations of cell specifications. The first indicates a median-strength D flop cell and the latter a low-strength sequential cell.

```
<spirit:cellSpecification>
   <spirit:cellFunction>dff</spirit:cellFunction>
</spirit:cellSpecification>
<spirit:cellSpecification>
   <spirit:cellClass spirit:strength="low">sequential</spirit:cellClass>
</spirit:cellSpecification>
```

7.11.15 Other clock drivers

7.11.15.1 Schema

The following schema defines the information contained in the **otherClockDrivers** element, which may appear within a **component** element.



7.11.15.2 Description

The **otherClockDrivers** element defines clocks within a component that are not directly associated with a top-level port, e.g., virtual clocks and generated clocks. The **otherClockDrivers** element contains one or more **otherClockDriver** elements, each of which represents a single clock. The **otherClockDriver** element consists of a number of sub-elements which define the format of the clock waveform.

a) **clockPeriod**, **clockPulseOffset**, **clockPulseValue** and **clockPulseDuration** (all required) are all detailed in the description of the element **clockDriver**. See 7.11.7.

b) **clockName** (mandatory) attribute indicating the name of the clock for reference by a constraint. The **clockName** element is of type ***Name***.

c) **clockSource** (optional) attribute defines the physical path and name to the clock generation cell.

7.11.15.3 Example

This example shows a virtual and a generated clock within the **otherClockDrivers** element.

```
<spirit:otherClockDrivers>
   <spirit:otherClockDriver spirit:clockName="virtClock">
      <spirit:clockPeriod>5</spirit:clockPeriod>
      <spirit:clockPulsOffset>0</spirit:clockPulseOffset>
```

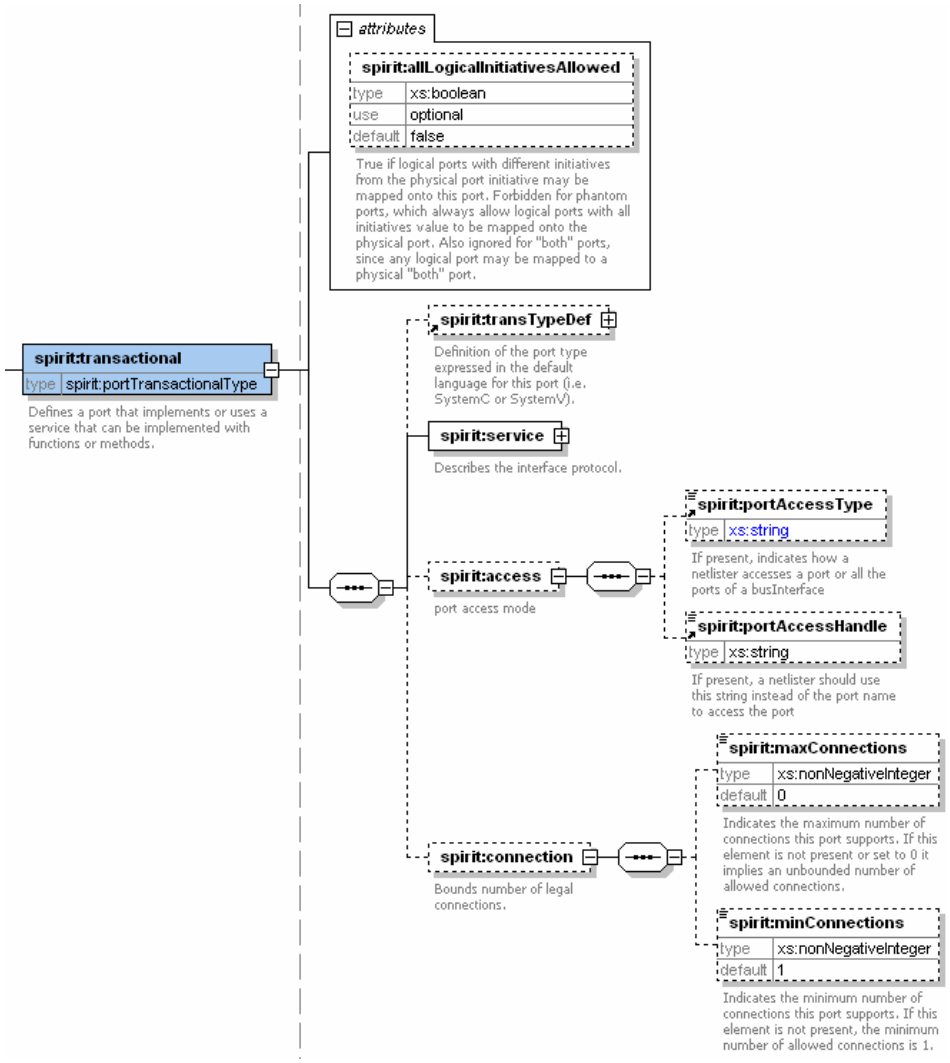
```
            <spirit:clockPulseValue>1</spirit:clockPulseValue>
            <spirit:clockPulseDuration>2.5</spirit:clockPulseDuration>
         </spirit:otherClockDriver>
         <spirit:otherClockDriver spirit:clockName="genClock"
            spirit:clockSource="i_clkGen/clk1">
            <spirit:clockPeriod spirit:units="ps">10</spirit:clockPeriod>
            <spirit:clockPulsOffset spirit:units="ps">2</spirit:clockPulseOffset>
            <spirit:clockPulseValue>0</spirit:clockPulseValue>
            <spirit:clockPulseDuration spirit:units="ps">5
               </spirit:clockPulseDuration>
         </spirit:otherClockDriver>
      <spirit:otherClockDrivers>
```

7.11.16 Transactional ports

7.11.16.1 Component transactional port type

7.11.16.1.1 Schema

The following schema defines the information contained in the **transactional** element (in a **component**/**model**/**ports**/**port** element).



7.11.16.1.2 Description

A **transactional** element in a component model port enables to define a physical transactional port of the component, which implements or uses a service. A service can be implemented with functions or methods. It contains the following elements.

a) **allLogicalDirectionsAllowed** (optional) attribute defines the possible legal combinations for the initiative (defined in **service/initiative**, see 7.11.16.3) of ports between the component and the abstraction definition. See 6.2 on bus interfaces. The **allLogicalDirectionAllowed** attribute is of type ***Boolean***. If **true** logical ports with different initiatives from the physical port initiative may be mapped together. Forbidden for phantom ports, which always allow logical ports with all initiatives value to be mapped onto the physical port. Also ignored for "both" ports, since any logical port may be mapped to a physical "both" port.

b) **transTypeDef** (optional) defines the port type expressed in the default language for this port. See 7.11.16.2.

c) **service** (mandatory) describes the interface protocol associated to the transactional port. See 7.11.16.3.

d) **access** (optional) defines the access for a port.

1) **portAccessType** (optional) indicates to a netlister how to access the port. The portAccessType shall one of two possible values **ref** or **ptr**. If **ref** it should access theport directly and if **ptr** it should access the port with a pointer.

2) **portAccessHandle** (optional) indicates to a netlister the string to use to access the port, instead of the port name. The portAccessHandle is of type ***string***.

e) **connection** (optional) defines the number of legal connections for a port.

1) **maxConnections** (optional) indicating the maximum number of connections that this port supports. Its default value is 0, which indicates an unbounded number of legal connections. The **maxConnections** element is of type ***nonNegativeInteger***.

2) **minConnections** (optional) indicating the minimum number of connections that this ports supports. Its default value is 1. The **minConnections** element is of type ***nonNegativeInteger***.

7.11.16.1.3 Example

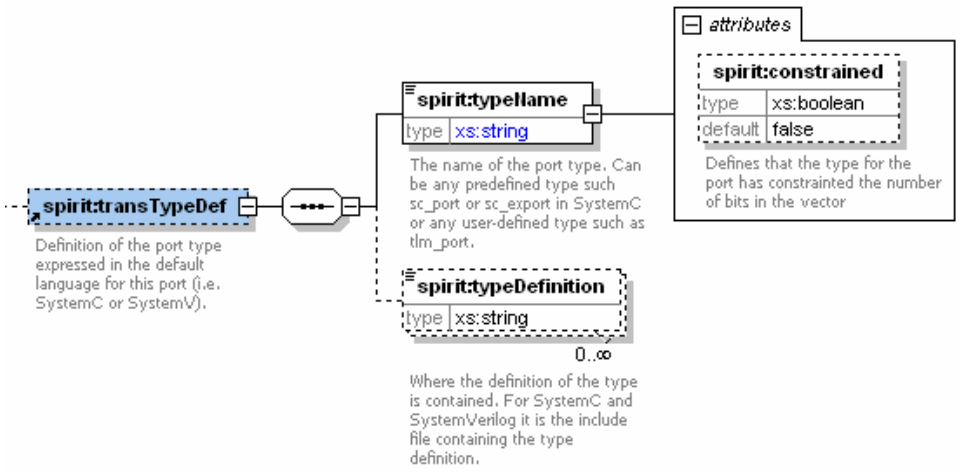
The following example shows the transactional type definition of a custom specific `tlm_port`, defined in the include file `tlm_port.h`.

```
<spirit:transTypeDef>
    <spirit:typeName>tlm_port</spirit:typeName>
    <spirit:typeDefinition>tlm_port.h</spirit:typeDefinition>
</spirit:transTypeDef>
```

7.11.16.2 Component transactional port type definition

7.11.16.2.1 Schema

The following schema defines the information contained in the **transTypeDef** element (in a **component**/**model**/**ports**/**port**/**transactional** element).



7.11.16.2.2 Description

A **transTypeDef** element defines the port type expressed in the default language for this port (e.g., SystemC or SystemVerilog). It contains the following elements.

a) **typeName** (mandatory) defines the port type (such as `sc_port`/`sc_export` in SystemC or any user-defined type, such as `tlm_port`). The **typeName** element may be associated with an optional

Boolean **constrained** attribute (the default value is **false**). If **true** this indicates that the port type definition has constrained the number of bits in the vector.

b) **typeDefinition** (optional) indicates a location where the type is defined,e.g., in SystemC and SystemVerilog, this is the include file containing the type definition. The **typeDefinition** element is of type ***string***.

7.11.16.2.3 Example

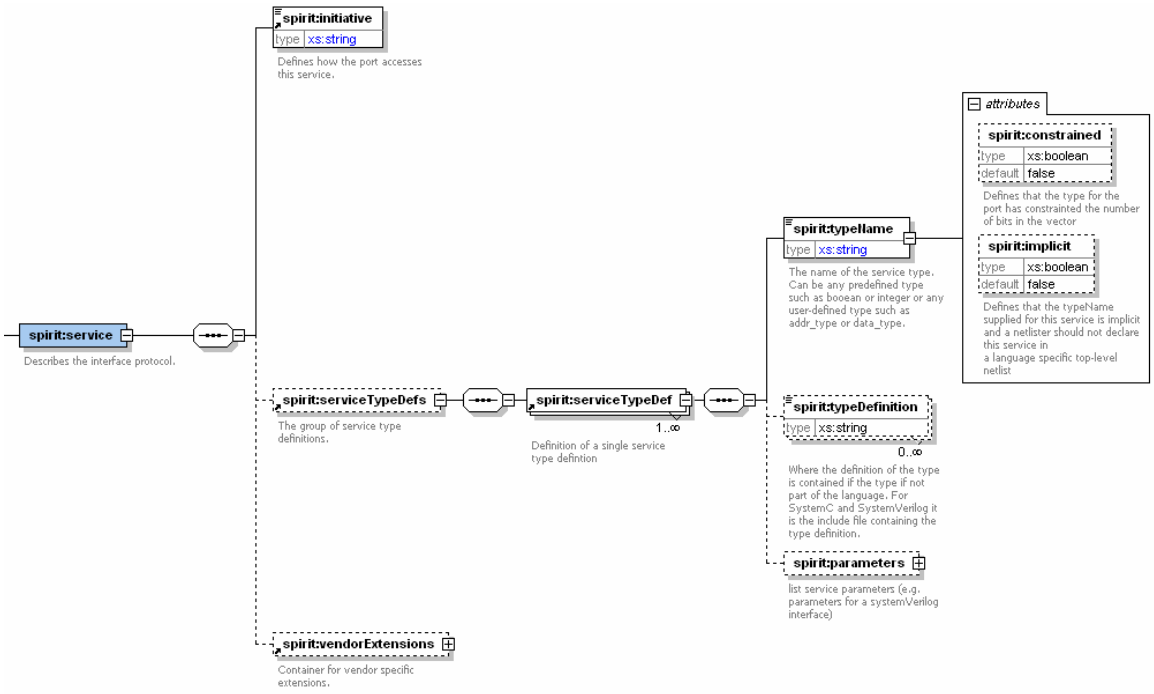
The following example shows the transactional type definition of a custom specific `tlm_port`, defined in the include file `tlm_port.h`.

```
<spirit:transTypeDef>
    <spirit:typeName>tlm_port</spirit:typeName>
    <spirit:typeDefinition>tlm_port.h</spirit:typeDefinition>
</spirit:transTypeDef>
```

7.11.16.3 Component transactional port service

7.11.16.3.1 Schema

The following schema defines the information contained in the **service** element (in a **component**/**model**/**ports**/**port**/**transactional** element).



7.11.16.3.2 Description

A **service** element describes the interface protocol associated to the transactional port. It contains the following elements and attributes.

a) **initiative** (mandatory) defines the type of access: **requires**, **provides**, **both**, or **phantom**.

1) For example, a SystemC `sc_port` should be defined with the **requires** initiative, since it requires a SystemC interface. A SystemC `sc_export` should be defined with the **provides** initiative, since it provides a SystemC interface.

2) A **both** value indicates the type of access is both **requires** and **provides**.

3) A **phantom** value indicates the type of access is a phantom port.

Phantom ports are additional ports in the component port list, which do not correspond to ports of the implementation. As with real component ports, the mapping of a set of logical bus ports to that phantom port implies any design using that component shall connect those logical ports with no intervening logic. The difference is a real component port needs to have a corresponding port in any RTL, TLM, or hierarchical IP-XACT implementation of the component; whereas, for phantom ports there is no corresponding port in the implementation. See 7.11.17.

b) **serviceTypeDefs** (optional) contains one or more **serviceTypeDef** elements. This **serviceTypeDef** element defines a single service type definition.

1) **typeName** (mandatory) defines the name of the service type (can be any predefined type, such as Boolean or any user-defined type, such as `addr_type`). The **typeName** element may be defined with two optional attributes: **constrained** (a Boolean indicating if the port type has constrained the number of bits in the vector) and **implicit** (a Boolean indicating a netlister should not declare this service in a language-specific, top-level netlist).

2) **typeDefinition** (optional) indicates a location where the type is defined,e.g., in SystemC and SystemVerilog, this is the include file containing the type definition.

3) **parameters** (optional) specifies any service type parameters. See X.Y.Z.

c) **vendorExtensions** (optional) adds any extra vendor-specific data related to the service.

7.11.16.3.3 Example

The following example shows the definition of the service provided by a SystemC port.

```
sc_export< pvt_if<ADDR, DATA> > pvt_port
```

```
<spirit:service>
   <spirit:initiative>provides</spirit:initiative>
   <spirit:serviceTypeDefs>
      <spirit:serviceTypeDef>
         <spirit:typeName>pvt_if</spirit:typeName>
         <spirit:parameters>
            <spirit:parameter name="addr" resolve="user">ADDR
               </spirit:parameter>
            <spirit:parameter name="data" resolve="user">DATA
               </spirit:parameter>
         </spirit:parameters>
      </spirit:serviceTypeDef>
   </spirit:serviceTypeDefs>
</spirit:service>
```

7.11.17 Phantom ports

In some components, the RTL or TLM implementation of the component does not fully implement the functionality of the component described by IP-XACT. In RTL components, this is typically because the component has to work in design flows that only allow a signal to be routed though an RTL component if there is some logic within the RTL component associated with that signal. This is particularly a problem for components containing channels.

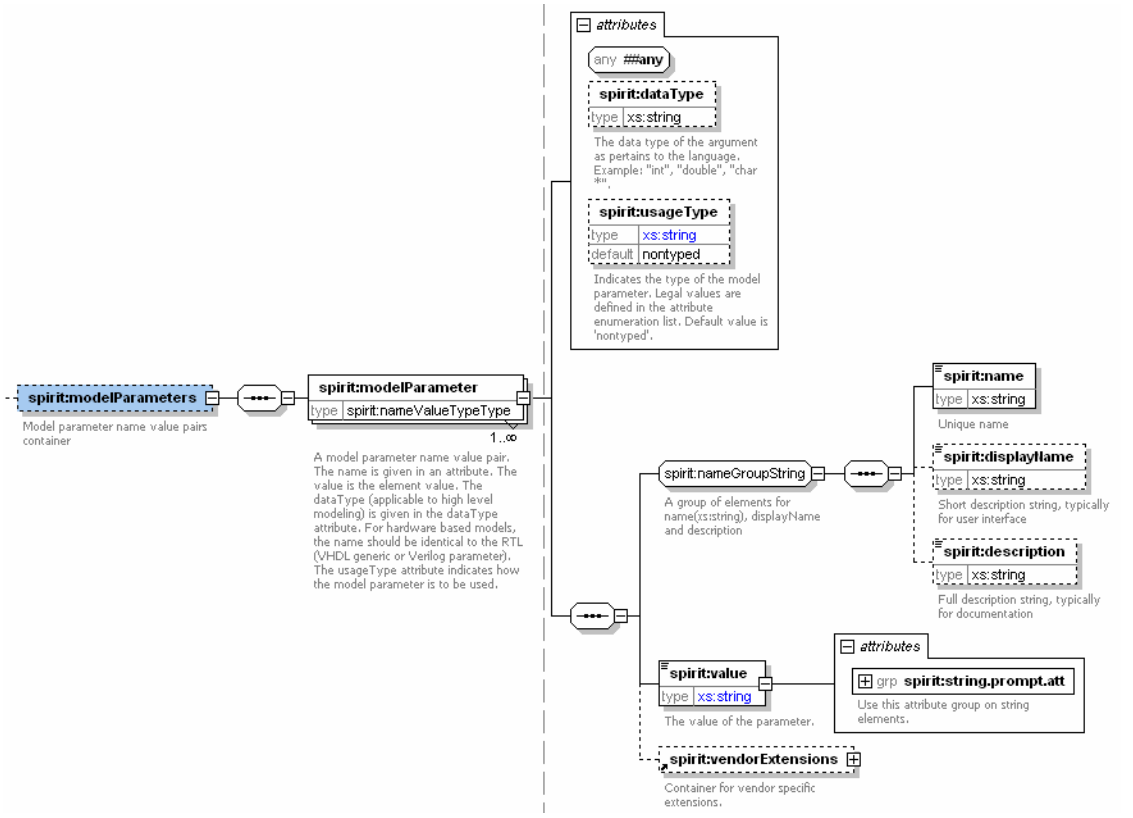
An IP-XACT channel is supposed to represent the complete bus infrastructure between the master, slave, and system bus interfaces connected to the bus. As such, the component containing the channel should contain everything that is needed to create this infrastructure. In many buses, however, some signals are directly connected between the components attached to the bus, with no intervening logic. This is most often the case with clock and reset signals. If the component is to be usable in a wide range of design flows these signals cannot be included in the RTL of the component.

To fully describe such a channel component and allow netlisters that have no special knowledge of that bus type to netlist designs containing it, IP-XACT describes these additional connections as phantom ports. *Phantom ports* are additional ports included in the component's port list, but marked as **phantom**. As with real component ports, the mapping of a set of logical bus ports to that phantom port implies any design using that component needs to connect those logical ports with no intervening logic. The difference is a real component port needs to have a corresponding port in any RTL, TLM, or hierarchical IP-XACT implementation of the component; whereas, for phantom ports there is no corresponding port in the implementation.

7.11.18 modelParameters

7.11.18.1 Schema

The following schema details the information contained in the **modelParameters** element, which may appear as an element inside the top-level **component**/**model** or **abstractor**/**model** element.



7.11.18.2 Description

Model parameters are most often used in HDL languages to specify information that is passed to the model to configure it for a process. The **modelParameters** element may contain any number **modelParameter** elements. The **modelParameter** elements describe the properties for a single parameter that is applied to all the models specified under the **model**/**views** element. It contains the following elements.

a) **dataType** (optional) attribute specifies the data type as it pertains to the language of the model. This definition is used to define the type for component declaration and such and has no semantic meaning. For example, systemC this could be `int`, `double`, `char*`, etc. For VHDL this could be std_logic, std_logic_vector, integer, etc.

b) **usageType** (optional) attribute specifies how this parameter is used in different modeling languages: **nontyped** (the default) and **typed**. See 7.11.18.2.1.

a) ***nameGroup*** group includes the following. See X.Y.Z .

1) **name** (mandatory) identifies the **modelParameter**.

2) **displayName** (optional) allows a short descriptive text to be associated with the register.

3) **description** (optional) allows a textual description of the register.

b) **value** (mandatory) contains the actual value of the parameter. The **value** element is of type ***string***. The **value** element is configurable with attributes from ***string.prompt.att***, see X.Y.Z on configuration.

c) **vendorExtensions** (optional) adds any extra vendor-specific data related to the **modelParamter**.

See also: All SCRs that apply to the **parameter** element also apply to **modelParameters**, see Table B5.

7.11.18.2.1 Typed and non-typed parameters classification

There are two categories of parameters: type and non-typed.

The *type* parameters (or declaration parameters) appear in object-oriented (OO) languages such a C++/ SystemC or SystemVerilog.

In C++/SystemC, these are named `Class` template parameters. Templates can be used to develop a generic class prototype (specification) which can be instantiated with different data types. This is very useful when the same kind of class is used with different data types for individual members of the class. Parameterized types are used as data types and then a class can be instantiated, i.e., constructed and used by providing arguments for the parameters of the class template. A class template is a specification of how a class should be built (i.e., instantiated) given the data type or values of its parameters.

`Class` template parameters can have default arguments, which are used during class template instantiation when arguments are not provided. Because the provided arguments are used starting from the far left parameter, default arguments should be provided for the right-most parameters.

Example 1

```
template <typename T>
class FIFO {
          FIFO();
          T pull();
          void push(T &x);
};
```

In SystemVerilog, typed parameters are named type parameters. Type parameters can be used in SystemVerilog classes, interfaces, or modules to provide the basic function of C++ templates.

Example 2

```
typedef bit[32] DataT;
interface FIFO #(type T);
          Method T pull();
          Method push (T x);
endinterface: FIFO
```

The generic *non-typed* parameters (or initialization parameters) appear in all languages (procedural or OO) and in particular in VHDL, Verilog, SystemC, and SystemVerilog. A non-typed parameter is like an ordinary (function-parameter) declaration. In SystemC, it represents a constant in a class template definition or a parameter in a class constructor, i.e., this can be determined at compilation time. In VHDL, it is represented by generics. In Verilog or SystemVerilog, it is represented by parameters.

Example 3

Here is an example of non-typed parameters usage on a simple GCD model expressed in various languages.

VHDL

```
entity GCD is
generic (Width: natural);
port (
Clock,Reset,Load: in std_logic;
                    A,B:      in unsigned(Width-1 downto 0);
                    Done:     out std_logic;
                    Y:        out unsigned(Width-1 downto 0));
end entity GCD;
```

(System)Verilog

```
module GCD (Clock, Reset, Load, A, B, Done, Y);
parameter Width = 8;
                    input     Clock, Reset, Load;
                    input     [Width-1:0] A, B;
                    output    Done;
                    output    [Width-1:0] Y;
…
endmodule
```

SystemC

```
template <unsigned int Width = 8>
SC_MODULE (GCD) {
                    sc_in<bool> Clock, Reset, Load;
                    sc_in<sc_uint<Width> >a, b;
                    sc_out<bool> Done;
                    sc_out<sc_uint<Width> > y;
                    …
}
```

These two kinds of parameters (typed and non-typed) can be combined to model complex IP modules.

Example 4

In SystemC:

```
template <typename T> // type parameter
class testModule : public sc_module {
public:
                    testModule(sc_module_name modnamemodname, string
   portname) :
                       // non type parameters
                       sc_module(modname),
                       testport(portname) {…}
                       sc_port<T> testport;
};
```

In a top SC netlist design, such a class is instantiated as follows.

```
testModule<bool> test("myModuleName","port1");
```

In IP-XACT, the **testModule** parameters are represented as follows.

```
<spirit:modelParameters>
   <!-- template parameter -->
   <spirit:modelParameter spirit:usageType="typed">
      <spirit:name>T</spirit:name>
      <spirit:value
         spirit:choiceRef="typenameChoice"
         spirit:configGroups="requiredConfig"
         spirit:id="Tid"
         spirit:prompt="T:"
         spirit:resolve="user">boolean</spirit:value>
   </spirit:modelParameter>
   <!-- constructor parameters -->
   <spirit:modelParameter spirit:usageType="nontyped">
      <spirit:name>modname</spirit:name>
      <spirit:value
         spirit:choiceRef="modulenameChoice"
         spirit:configGroups="requiredConfig"
         spirit:id="modnameId"
         spirit:prompt="moduleName:"
         spirit:resolve="user">myModuleName</spirit:value>
   </spirit:modelParameter>
   <spirit:modelParameter spirit:usageType="nontyped">
      <spirit:name>portname</spirit:name>
      <spirit:value
         spirit:choiceRef="portnameChoice"
         spirit:configGroups="requiredConfig"
         spirit:id="portnameid"
         spirit:prompt="portName:"
         spirit:resolve="user">port1</spirit:value>
   </spirit:modelParameter>
</spirit:modelParameters>
```

7.11.18.2.2 Generic parameters mapping in different languages

Table 7 summarizes the two kind of parameters (initialization and declaration) expressed in the four most commonly used HW languages.

Table 7—Parameter mappings

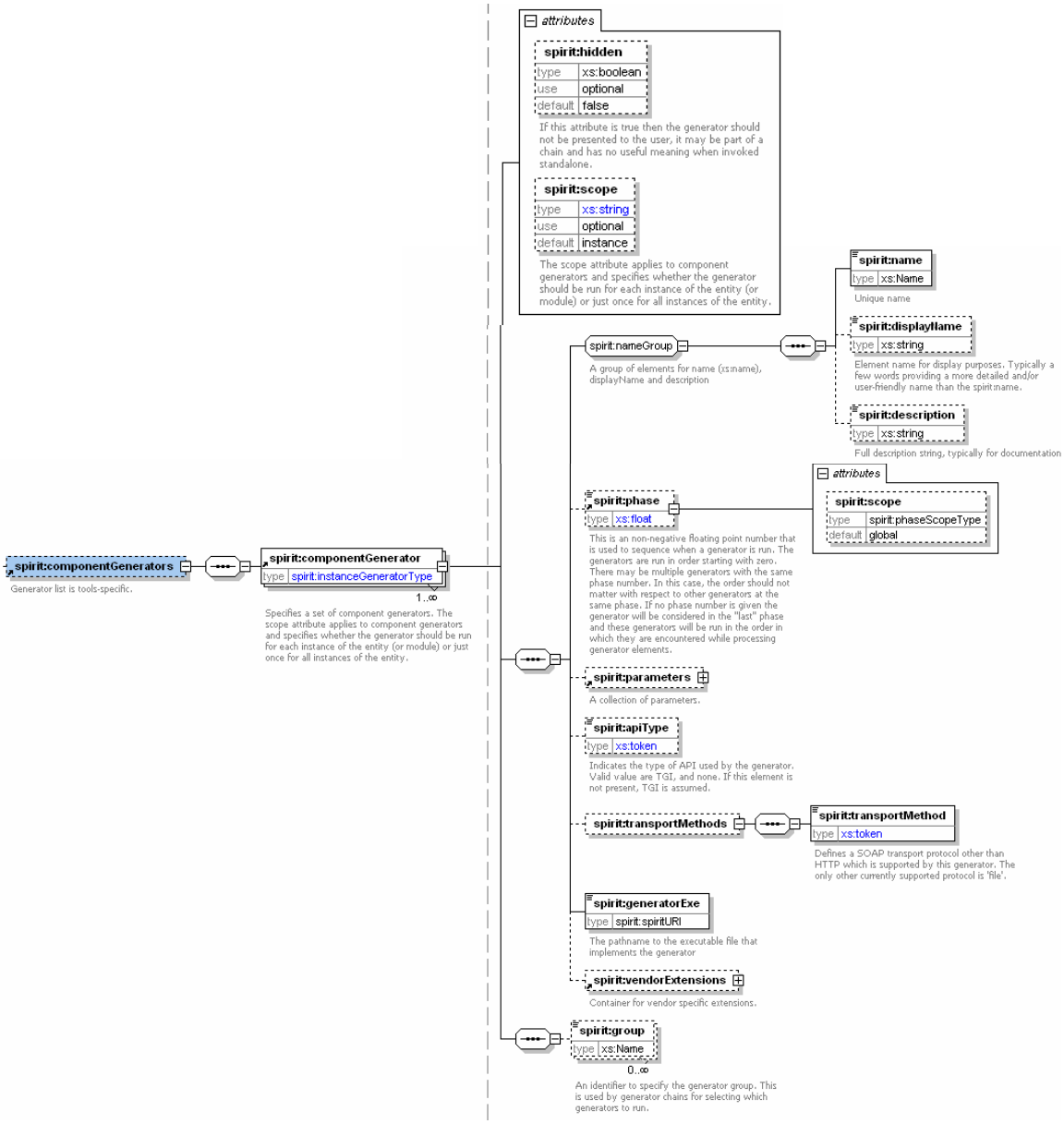
Language	Non-typed parameters (initialization)	Typed parameters (declaration)
VHDL	generics	N.A
Verilog	parameter	N.A
SystemC	constructor	Template (constant or variable type)
SystemVerilog	parameter	parameter

A *declaration parameter* (e.g., `int`) shall be used when declaring an IP instance in a top netlist (e.g,. `myIP int myIntIP;`). An *initialization parameter* (e.g., `myName`) shall be used when initializing the instance of that IP (e.g., `myIntIP("myName");`).

7.12 Component generators

7.12.1 Schema

The following schema details the information contained in the **componentGenerators** element, which may appear as an element inside the top-level **component** element.



7.12.2 Description

The **componentGenerators** element contains an unbounded list of **componentGenerator** elements. Each **componentGenerator** element defines a generator that are assigned and may be run on this component. **componentGenerator** contains two attributes: **hidden** and **scope**. The **hidden** (optional) attribute specifies, when ***True***, this generator shall not be run as the initial generator and is required to be run as port of a chain.

If ***False*** (the default), this generator may be run as an initial generator or in a generator chain. This attribute is of type ***Boolean***. The **scope** (optional) attribute is an enumerated list of **instance** and **entity**. **instance** indicates this generator shall be run once for all instances of this component. **entity** indicates this generator shall be run once for each instance of this component.

componentGenerator contains the following elements.

a) ***nameGroup*** group includes the following.

   1) **name** (mandatory) identifies the component generator. The **name** element is of type ***Name***.

   2) **displayName** (optional) allows a short descriptive text to be associated with the component generator. The **displayName** element is of type ***string***.

   3) **description** (optional) allows a textual description of the component generator. The **description** element is of type ***string***.

b) **phase** (optional) determines the sequence in which a generators are run. Multiply selected generators are run in order starting with zero (`0`). If generators have the same phase numbers, the order shall be interpreted as not important and the generators can be run in any order. If no phase number is given the generator is considered in the "last" phase and these generators are run in the order they are encountered while processing **componentGenerator** elements. The **phase** element is of type ***float*** and shall also be a positive number.

   phase can also contain a **scope** (optional) attribute specifying the scope of the phase number in this generator as related to other generators. This is an enumerated list of **global** or **local**. **global** (the default) indicates the phase number shall be used across all generators when determining the generator sequence. **local** indicates the phase number shall only be used in comparison with generators defined within this component description.

c) **parameters** (optional) specifies any **componentGenerator** type parameters. See X.Y.Z.

d) **apiType** (optional) indicates the type of API used by the generator: an enumerated list of **TGI** or **none**. **TGI** indicates the generator uses communication to the design environment compliant with the TGI. **none** indicates the generator does not use any communication with the DE.

e) **transportMethods** (optional) defines alternate SOAP transport protocol that this generator can support. The default SOAP transport protocol is HTTP if this element is not present.

   transportMethod specifies the alternate transport protocol. This element is an enumerated list of only one element **file**. **file** indicates the SOAP transport protocol is transported to the DE view of a file or file handle.

f) **generatorExe** (mandatory) contains an absolute or relative (to the location of the containing description) path to the generator executable. The path may also contain environment variables from the host system, which are used to abstract the location of the generator. The **generatorExe** element is of type ***spiritURI***.

g) **vendorExtensions** (optional) adds any extra vendor-specific data related to the **componentGenerator**.

h) **group** (optional, unbounded) is a list of names used to assign this generator to a group with other generators. These **group** names are then referenced by a generator chain selector to forming a chain of generators. See X.Y.Z. The **group** element is of type ***Name***.

7.12.3 Example

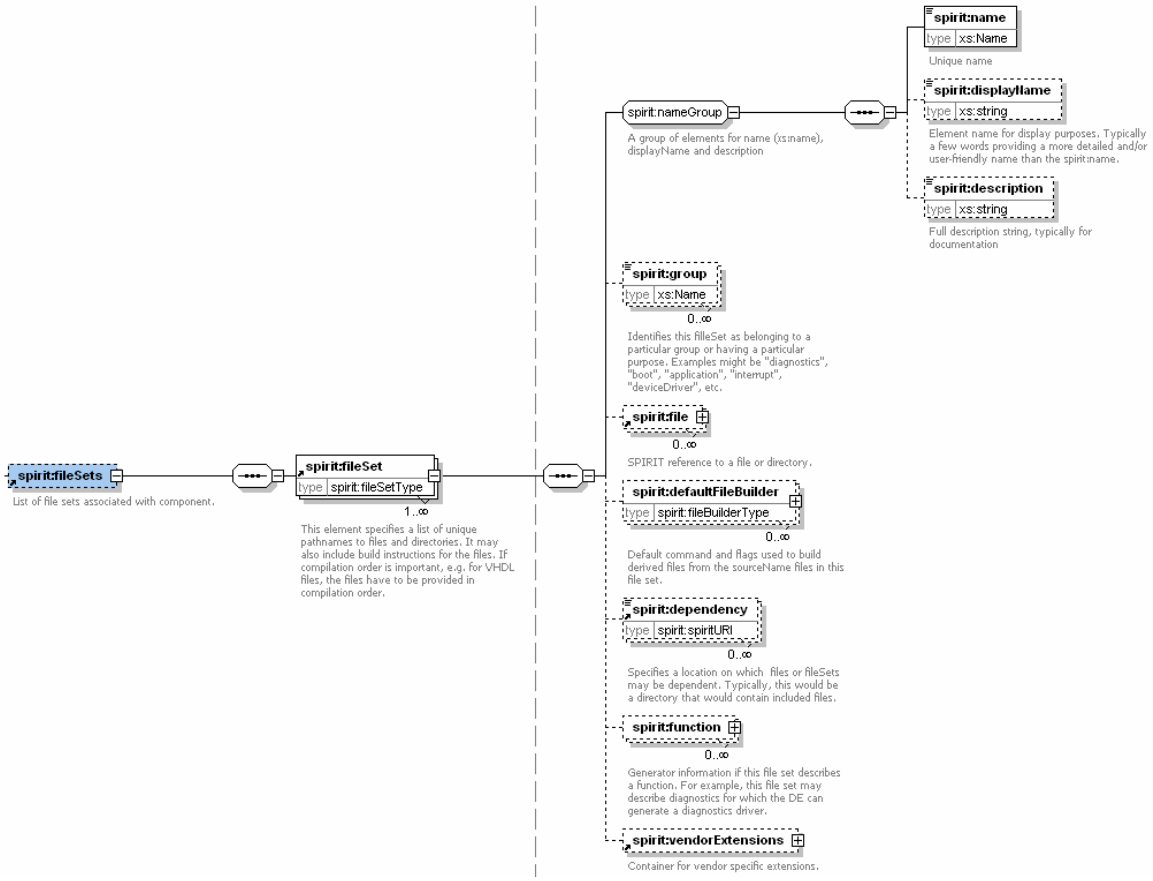
This example shows

7.13 Files

7.13.1 filesets

7.13.1.1 Schema

The following schema details the information contained in the **fileSets** element, which may appear in component or an abstractor.



7.13.1.2 Description

The **fileSets** element contains may contain one or more **fileSet** elements. A **fileSet** contains a list of files associated with a component. A **Fileset** can establish the (relative) path directory of files and elements associated with a component and/or include any build instructions. If completion order is important (e.g., for VHDL files), the files shall be listed in the order needed for completion. **fileSet** has the following mandatory and optional elements.

a) ***nameGroup*** group includes the following. See X.Y.Z.

   1) **name** (mandatory) identifies the file set. The **name** element is of type ***Name***.

   2) **displayName** (optional) allows a short descriptive text to be associated with the file set. The **displayName** element is of type ***string***.

   3) **description** (optional) allows a textual description of the file set. The **description** element is of type ***string***.

b) **group** (optional, unbounded) describes the function or purpose of the file set with a single word group name (e.g., `diagnostics`, `interrupt`, etc.). The **group** element is of type ***Name***.

c) **file** (optional, unbounded) references a single file or directory associated with the file set (see 7.13.2).

d) **defaultFileBuilder** (optional, unbounded) specifies the default build commands for the files within this file set.

e) **dependency** (optional, unbounded) is the path to a directory containing (include) files on which the file set depends. The **dependency** element is of type ***spiritURI***.

f) **function** (optional, unbounded) specifies the information about a function for a generator (see 7.13.5).

g) **vendorExtensions** (optional) provides a place for any vendor-specific extensions.

7.13.1.3 Example

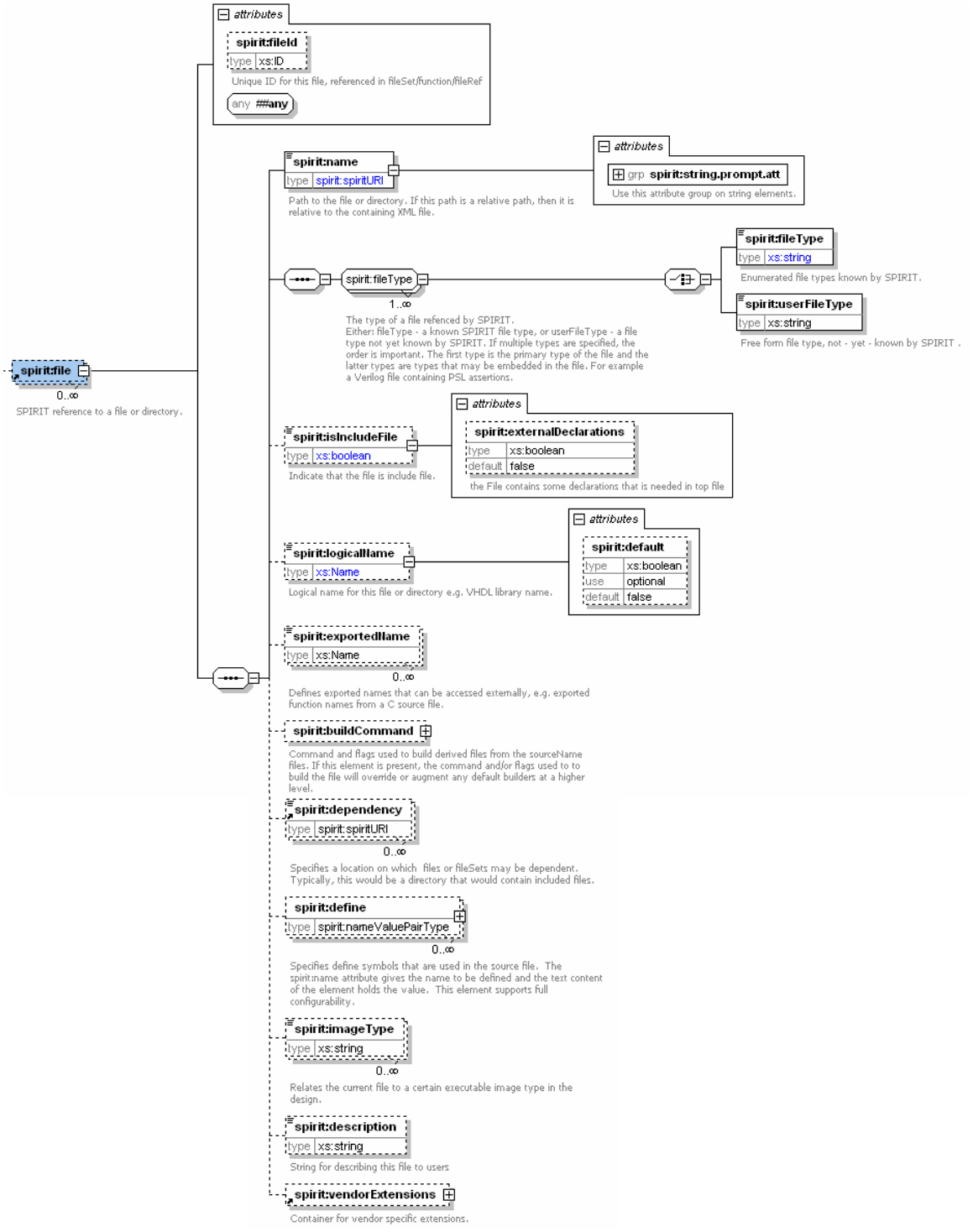
The following is an example of a **fileSet** with two VHDL files.

```
<spirit:fileSets>
   <spirit:fileSet spirit:fileSetId="fs-vhdlSource">
      <spirit:name>fs-vhdlSource</spirit:name>
      <spirit:file>
         <spirit:name>hdlsrc/timers.vhd</spirit:name>
         <spirit:fileType>vhdlSource</spirit:fileType>
         <spirit:logicalName>leon2_timers</spirit:logicalName>
      </spirit:file>
      <spirit:file>
         <spirit:name>hdlsrc/leon2_Timers.vhd</spirit:name>
         <spirit:fileType>vhdlSource</spirit:fileType>
         <spirit:logicalName>leon2_timers</spirit:logicalName>
      </spirit:file>
   </spirit:fileSet>
</spirit:fileSets>
```

7.13.2 file

7.13.2.1 Schema

The following schema details the information contained in the **file** element, which may appear as an element inside the **fileset** element.



7.13.2.2 Description

A **file** is a reference to a file or directory. It is an optional element of a **fileset**. If completion order is important (e.g., for VHDL files), the files shall be listed in the order needed for completion. The **file** element

contains an attribute **fileId** (optional) which is used for references to this file from inside **fileSet**/**function**/**fileRef** element. The **file** element also allows for vendor attributes to be applied. **file** contains the following elements.

a) **name** (mandatory) contains an absolute or relative (to the location of the containing description) path to a file name or directory. The path may also contain environment variables from the host system, used to abstract the location of files. The **name** element is of type ***spiritURI***. The **name** element is configurable with attributes from ***string.prompt.att***, see X.Y.Z on configuration.

b) ***fileType*** (required, unbounded) group contains one of the following two elements.

1) **fileType** (mandatory) describes the type of file referenced from the this enumerated list of industry standard files: **unknown**, **cSource**, **cppSource**, **asmSource**, **vhdlSource**, **vhdlSource-87**, **vhdlSource-93**, **verilogSource**, **verilogSource-95**, **verilogSource-2001**, **swObject**, **swObjectLibrary**, **vhdlBinaryLibrary**, **verilogBinaryLibrary**, **unelaboratedHdl**, **executableHdl**, **systemVerilogSource**, **systemVerilogSource-3.0**, **systemVerilogSource-3.1**, **systemCSource**, **systemCSource-2.0**, **systemCSource-2.0.1**, **systemCSource-2.1**, **veraSource**, **eSource**, **perlSource**, **tclSource**, **OVASource**, **SVASource**, **pslSource**, **systemVerilogSource-3.1a**, and **SDC**.

2) **userFileType** (mandatory) describes any other file type that can not be described from the list for **fileType**. The **userFileType** element is of type ***string***.

c) **includeFile** (optional) when ***True***, declares the file as an include file. If this element is not present the default value is ***False***. **includeFile** is of type ***Boolean***. **includeFile** has an attribute **externalDeclarations** (optional), when ***True***, this indicates the include file is needed by users of any files in this file set. The default is **false**.

d) **logicalName** (optional) is the logical name for the file or directory, such as a VHDL library. The **logicalName** element is of type ***Name***. **logicalName** includes an attribute **default** (optional) that means something. The **default** attribute is of type ***Boolean*** and the default is **false**.

e) **exportedName** (optional, unbounded) defines any names that can be referenced externally. **exportedName** is of type ***Name***.

f) **buildCommand** (optional) contains flags or commands for building the containing source file. These flags or commands override any flags or commands present in higher-level **defaultFileBuilder** elements. See X.Y.Z.

g) **dependency** (optional, unbounded) is the path to a directory containing (include) files on which the file depends. The **dependency** element is of type ***spiritURI***.

h) **define** (optional, unbounded) specifies the define symbols to use in the source file. See 7.13.4.

i) **imageType** (optional, unbounded) relates the current file to an executable image type in the design.

j) **description** (optional) details the file for the user. The **description** element is of type ***string***.

k) **vendorExtensions** (optional) provides a place for any vendor-specific extensions.

See also SCR 12.1.

7.13.2.3 Example

The following is an example of two file sets. One with a Verilog file with a dependency on a directory and one with a VHDL file.

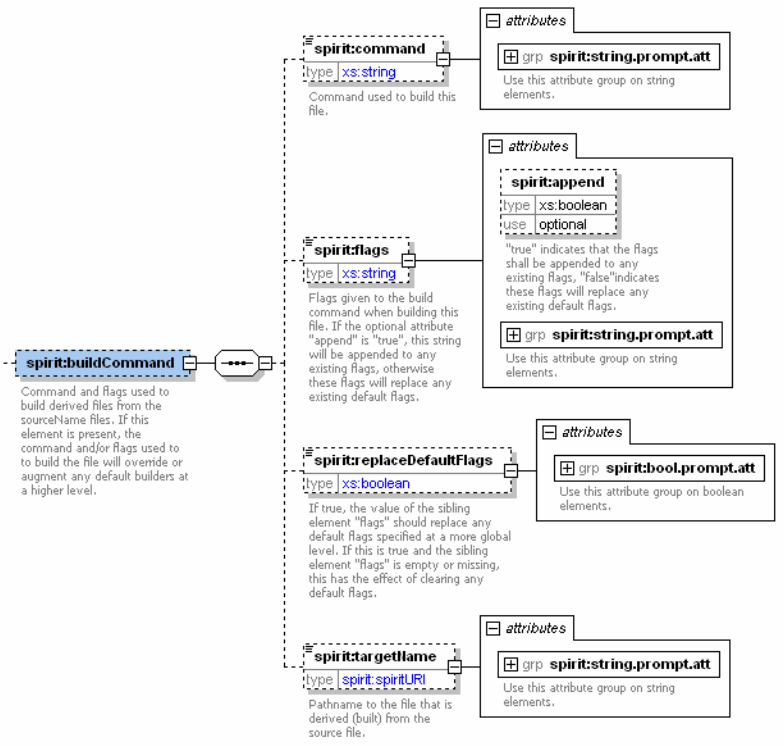
```
<spirit:fileSets>
    <spirit:fileSet>
      <spirit:name>fs-verilogSource</spirit:name>
       <spirit:file>
         <spirit:name>data/i2c/RTL/i2c.v</spirit:name>
         <spirit:fileType>verilogSource</spirit:fileType>
         <spirit:logicalName>i2c_lib</spirit:logicalName>
```

```
      </spirit:file>
      <spirit:dependency>data/i2c/RTL</spirit:dependency>
    </spirit:fileSet>
    <spirit:fileSet>
      <spirit:name>fs-vhdlWrapper</spirit:name>
      <spirit:file>
        <spirit:name>data/i2c/RTL/i2c.vhd</spirit:name>
        <spirit:fileType>vhdlSource</spirit:fileType>
        <spirit:logicalName>i2c_lib</spirit:logicalName>
      </spirit:file>
    </spirit:fileSet>
  </spirit:fileSets>
```

7.13.3 buildCommand

7.13.3.1 Schema

The following schema details the information contained in the **buildCommand** element, which may appear as an element inside the **file** element.



7.13.3.2 Description

A **buildCommand** contains flags or commands for building the containing source file. These flags or commands override any flags or commands present in higher-level **defaultFileBuilder** elements.

a) **command** (optional) element defines a compiler or assembler tool that processes the software of this type. The **command** element is of type ***string***. The **command** element is configurable with attributes from ***string.prompt.att***, see X.Y.Z on configuration.

b) **flags** (optional) documents any flags to be passed along with the software tool command. The **flag** element is of type ***string***. The **flags** element is configurable with attributes from ***string.prompt.att***, see X.Y.Z on configuration. The **flags** element contains an attribute **append** (optional), which, when ***True*** indicates the **flags** shall be appended to the current flags. If **false**, the **flags** shall replace the existing flags.

c) **replaceDefaultFlags** (optional) documents flags that replace any of the passed default flags. The **replaceDefaultFlags** element is of type ***Boolean***. The **replaceDefaultFlags** element is configurable with attributes from ***bool.prompt.att***, see X.Y.Z on configuration.

d) **targetName** (optional) defines the path to the file derived from the source file. The **targetName** element is of type ***spiritURI***. The **targetName** element is configurable with attributes from ***string.prompt.att***, see X.Y.Z on configuration.

7.13.3.3 Example

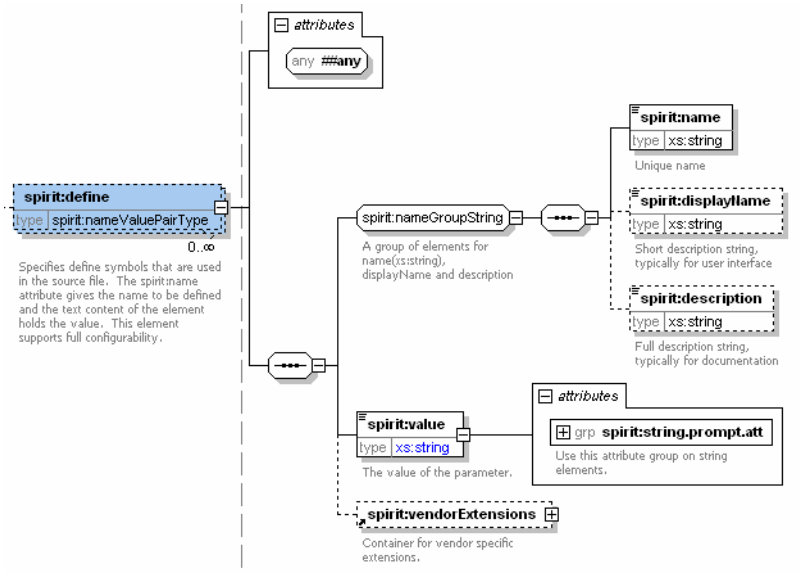
The following is an example.

```
<spirit:fileSets>
</spirit:fileSets>
```

7.13.4 define

7.13.4.1 Schema

The following schema details the information contained in the **define** element, which may appear as an element inside the **file** element.



7.13.4.2 Description

The **define** element specifies the define symbols to use in the source file. This **define** element allows for vendor attributes to be applied.

a) ***nameGroupString*** group includes the following. See X.Y.Z.

1) **name** (mandatory) identifies the name of the **define** symbol used in the source file. The **name** element is of type ***String***.
2) **displayName** (optional) allows a short descriptive text to be associated with the **define** element. The **displayName** element is of type ***string***.
3) **description** (optional) allows a textual description of the **define** element. The **description** element is of type ***string***.

b) **value** (mandatory) contains the value of the **define** symbol. The **value** element is of type ***string***. The **value** element is configurable with attributes from ***string.prompt.att***, see X.Y.Z on configuration.

c) **vendorExtensions** (optional) provides a place for any vendor-specific extensions.

7.13.4.3 Example

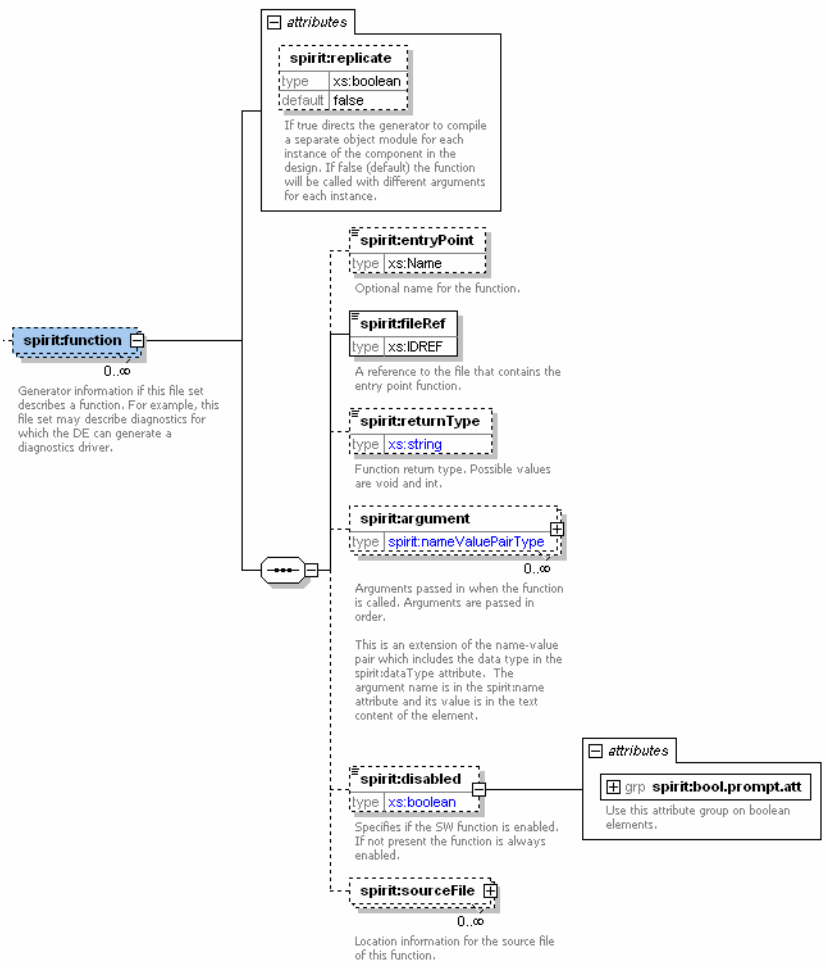
The following is an example

```
<spirit:fileSets>

</spirit:fileSets>
```

7.13.5 function

7.13.5.1 Schema

The following schema details the information contained in the **function** element, which may appear as an element inside the **fileset** element.



7.13.5.2 Description

A **function** specifies information about a generator function. **function** contains an attribute **replicate** (optional), when set to ***True***, the generator compiles a separate object module for each instance of the component in the design. This allows the function to be called with different attributes for each instance within the design (e.g., base address). The **replicate** attribute is of type ***Boolean*** and the default value is ***False***. **function** has the following elements.

a) **entryPoint** (optional) is the entry point name for the function or subroutine.

b) **fileRef** (mandatory) reference to the file that contains the entry point for the function. The value of this element shall match an attribute in **file**/**fileId**.

c) **returnType** (optional) is an enumerated ***string*** type which indicates the return type for the function. The two possible values are **int** and **void**.

d) **argument** (optional, unbounded) lists any arguments passed when this function is called. All arguments shall be passed in the order presented in this description. See 7.13.6.

e) **disabled** (optional) disables the software function. The **disabled** element is of type ***Boolean*** and the default is ***False***. When ***True***, the software function is not available for use. When ***False***, the function is available. The **disabled** element is configurable with attributes from ***bool.prompt.att***, see X.Y.Z on configuration.

f) **sourceFile** (optional, unbounded) references any source files. The order of the source files may be important, as this could indicate a compile order.

7.13.5.3 Example

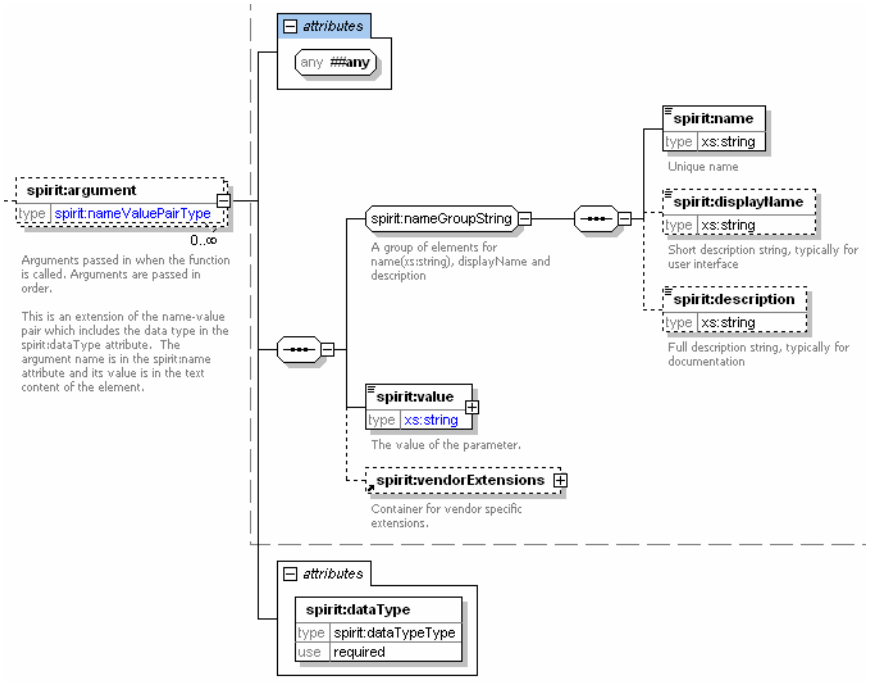
The following example includes a file with a `fileId` and a **function** referencing that file.

```
<spirit:fileSets>
        <spirit:fileSet spirit:fileSetId="fs-systemcSource">
            <spirit:name>sourceFiles</spirit:name>
            <spirit:file spirit:fileId="source">
                <spirit:name>src/source.cc</spirit:name>
                <spirit:fileType>systemCSource-2.1</spirit:fileType>
            </spirit:file>
            <spirit:function>
                <spirit:fileRef>source</spirit:fileRef>
                <spirit:returnType>void</spirit:returnType>
                <spirit:argument spirit:dataType="int">
                    <spirit:name>argument_1</spirit:name>
                    <spirit:value>0</spirit:value>
                </spirit:argument>
            </spirit:function>
        </spirit:fileSet>
    </spirit:fileSets>
```

7.13.6 argument

7.13.6.1 Schema

The following schema details the information contained in the **argument** element, which may appear as an element inside the **function** element.



7.13.6.2 Description

The **argument** element specifies the arguments passed to the **function** when making a call. All arguments shall be passed in the order presented in this description. The **dataType** (mandatory) attribute specifies the type for this argument, e.g., an `int` or `Boolean`. The **argument** element also allows for vendor attributes to be applied.

a) ***nameGroupString*** group includes the following. See X.Y.Z.

   1) **name** (mandatory) identifies the name of the **argument** in the **function**. The **name** element is of type ***String***.

   2) **displayName** (optional) allows a short descriptive text to be associated with the **argument**. The **displayName** element is of type ***string***.

   3) **description** (optional) allows a textual description of the **argument**. The **description** element is of type ***string***.

b) **value** (mandatory) contains the value of the **argument**. The **value** element is of type ***string***. The **value** element is configurable with attributes from ***string.prompt.att***, see X.Y.Z on configuration.

c) **vendorExtensions** (optional) provides a place for any vendor-specific extensions.

   sourceFile references any source files. Order is important in the source file. It has the following mandatory subelements.

   i) **sourceName** identifies the boot load file. Relative names are searched for in the project directory and the source of the component directory.

   ii) **fileType** references the SPIRIT file type. If multiple files are referenced, order is important. There are two categories that can be referenced:

   fileType includes file types and enumerated by SPIRIT and

   userFileType encompasses all other file types.

7.13.6.3 Example

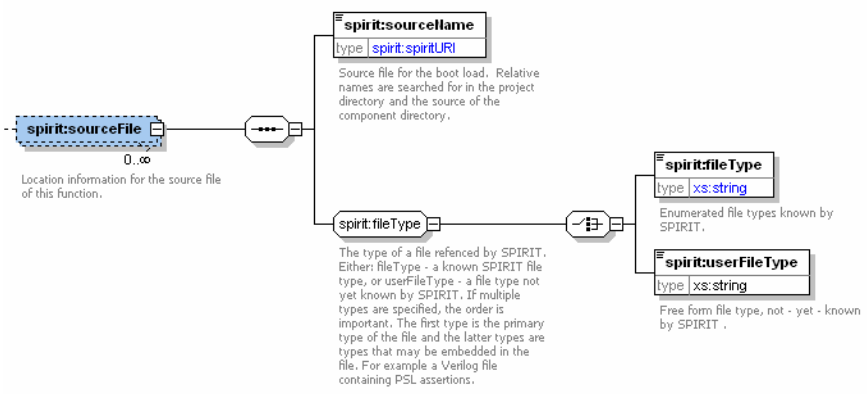
The following example includes a file with a `fileId` and a **function** referencing that file.

```
<spirit:fileSets>
        <spirit:fileSet spirit:fileSetId="fs-systemcSource">
            <spirit:name>sourceFiles</spirit:name>
            <spirit:file spirit:fileId="source">
                <spirit:name>src/source.cc</spirit:name>
                <spirit:fileType>systemCSource-2.1</spirit:fileType>
            </spirit:file>
            <spirit:function>
                <spirit:fileRef>source</spirit:fileRef>
                <spirit:returnType>void</spirit:returnType>
                <spirit:argument spirit:dataType="int">
                    <spirit:name>argument_1</spirit:name>
                    <spirit:value>0</spirit:value>
                </spirit:argument>
            </spirit:function>
        </spirit:fileSet>
    </spirit:fileSets>
```

7.13.7 sourceFile

7.13.7.1 Schema

The following schema details the information contained in the **sourceFile** element, which may appear as an element inside the **function** element.



7.13.7.2 Description

The **sourceFile** element specifies the location of the source files for this **function**. All source files shall be processed in the order presented in this description.

a) **sourceName** (mandatory) contains an absolute or relative (to the location of the containing description) path to a file name or directory. The path may also contain environment variables from the host system, used to abstract the location of files. Relative names are searched for in the project directory and the source of the component directory. The **sourceName** element is of type ***spiritURI***.

b) ***fileType*** (required) group contains one of the following two elements.

1) **fileType** (mandatory) describes the type of file referenced from the this enumerated list of industry standard files: **unknown**, **cSource**, **cppSource**, **asmSource**, **vhdlSource**, **vhdlSource-87**, **vhdlSource-93**, **verilogSource**, **verilogSource-95**, **verilogSource-2001**, **swObject**, **swObjectLibrary**, **vhdlBinaryLibrary**, **verilogBinaryLibrary**, **unelaboratedHdl**, **executableHdl**, **systemVerilogSource**, **systemVerilogSource-3.0**, **systemVerilogSource-3.1**, **systemCSource**, **systemCSource-2.0**, **systemCSource-2.0.1**, **systemCSource-2.1**, **veraSource**, **eSource**, **perlSource**, **tclSource**, **OVASource**, **SVASource**, **pslSource**, **systemVerilogSource-3.1a**, and **SDC**.

2) **userFileType** (mandatory) describes any other file type that can not be described from the list for **fileType**. The **userFileType** element is of type ***string***.

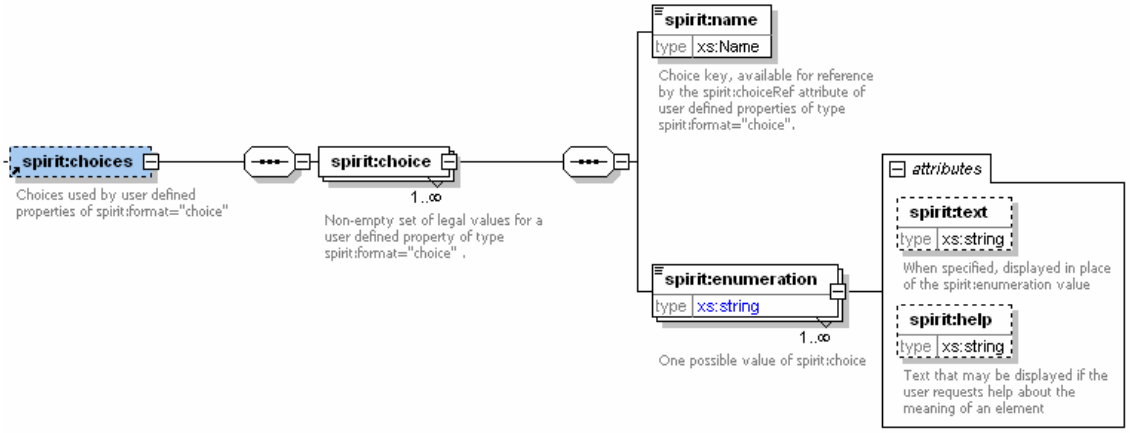
7.13.7.3 Example

The following example

7.14 Choices

7.14.1 Schema

The following schema details the information contained in the **choices** element, which may appear as an element inside the top-level **component** or **abstractor** element.



7.14.2 Description

The **choices** element contains an unbounded list of **choice** elements. Each **choice** element is a list of items used by a **modelParameter** element, **parameter** element, or any other configurable element with a **choiceRef** attribute. These elements indicate they are using a **choice** element by setting the attribute **choiceRef**. This **choiceRef** attribute shall reference a valid **choice**/**name** element in the containing XML document.

The **choice** definition contains the following elements.

a) **name** (mandatory) specifies the name of this list and is used by other element for reference. The **name** element is of type ***Name***.

b) **enumeration** (mandatory) is an unbounded list of of elements, where each holds a possible value that the referencing element may contain.

   1) **text** (optional) attribute causes optional text to be displayed when choosing the **choice** value. The resulting value stored in the configurable element corresponds to the enumeration value for the choice. If the **text** attribute is not present, the **enumeration** value may be displayed. The **text** element is of type ***string***.

   2) **help** (optional) attribute gives any additional information about this enumeration element.. The **help** element is of type ***string***.

See also: SCR 5.12.

7.14.3 Example

This example shows the addressable size (`width`) and the word size (`Dwidth`) of a memory component.

```
<spirit:model>
  <spirit:modelparameters>
```

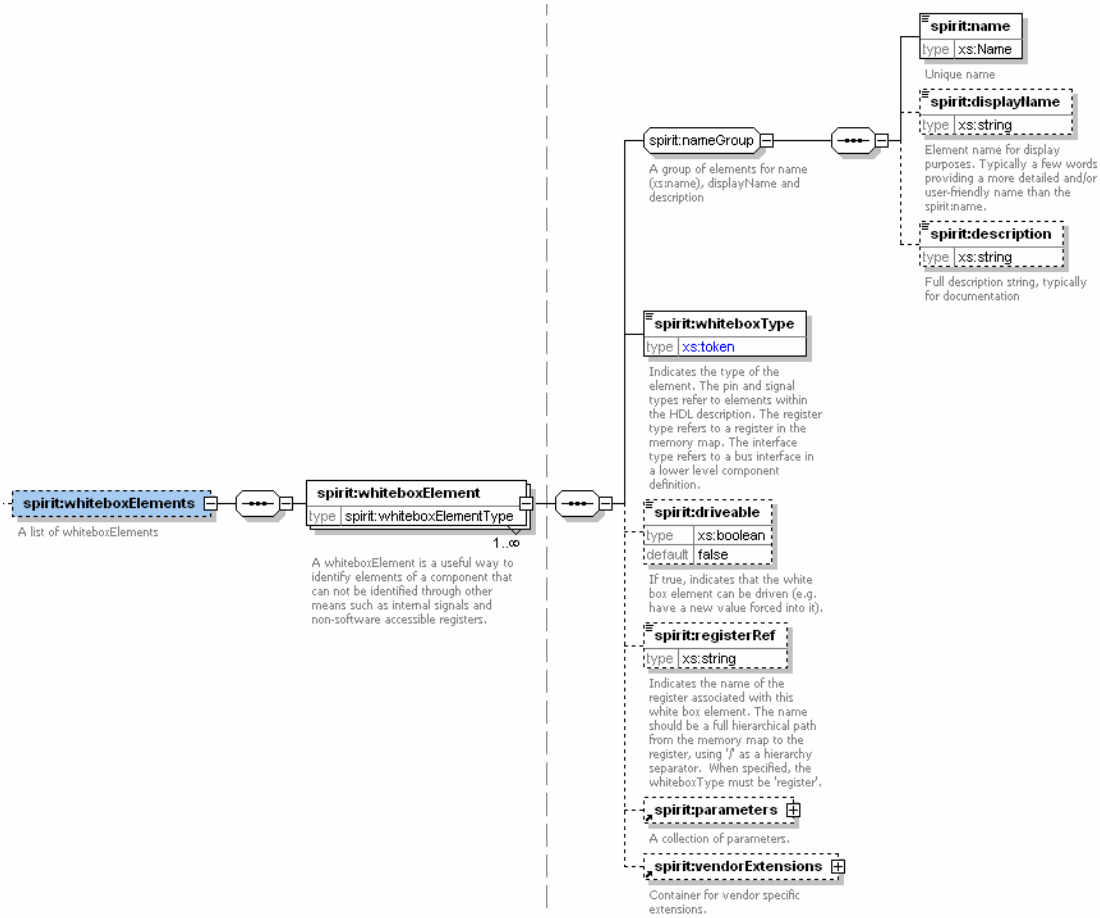
```
        <spirit:modelparameter spirit:name="width"
       spirit:choiceRef="widthOptions">1</spirit:modelparameter>
        <spirit:modelparameter spirit:name="Dwidth"
       spirit:choiceRef="DwidthOptions">4</spirit:modelparameter>
      </spirit:modelparameters>
    </spirit:model>

    <spirit:choices>
      <spirit:choice>
        <spirit:name>widthOptions</spirit:name>
        <spirit:enumeration spirit:text="8K">1</spirit:enumeration>
        <spirit:enumeration spirit:text="64K">2</spirit:enumeration>
        <spirit:enumeration spirit:text="256K">3</spirit:enumeration>
      </spirit:choice>
      <spirit:choice>
        <spirit:name>DwidthOptions</spirit:name>
        <spirit:enumeration spirit:text="2Bytes">4</spirit:enumeration>
        <spirit:enumeration spirit:text="4Bytes">5</spirit:enumeration>
        <spirit:enumeration spirit:text="8Bytes">6</spirit:enumeration>
      </spirit:choice>
    </spirit:choices>
```

7.15 Whitebox elements

Verification IP, such as monitors, have pseudo-physical bus interfaces to connect with bus interface ports under test, while not being an actual part of the design, but as part of a test bench instead. Other verification tools may require access to component IP in a design, at a level deeper than the interfaces defined for the component. A whitebox element provides such access. This can be used in situations where internal registers, flags, or whole IP-XACT interfaces need to be monitored or on internal nodes or interfaces driven by verification IP.

7.15.1 Schema

The following schema details the information contained in the **whiteboxElements** element, which may appear as an element inside the top-level **component** element.



7.15.2 Description

The **whiteboxElements** element contains the a list of one or more **whiteboxElement** elements. Each **whiteboxElement** element contains the following elements.

a) ***nameGroup*** group includes the following. See X.Y.Z.

   1) **name** (mandatory) identifies the whitebox element.

2) **displayName** (optional) allows a short descriptive text to be associated with the whitebox element.

3) **description** (optional) allows a textual description of the whitebox element.

b) **whiteboxType** (mandatory) documents this whitebox element's referent: a **register**, **pin**, **signal**, or **interface** within the component. **register** indicates that a register definition (referenced by the **registerRef** element) in this component can be mapped to physical signal(s) by a reference from the **model/view**. **pin** indicates a port on an internal instance in this component can be mapped to physical signal(s) by a reference from the **model/view**. **signal** indicates a signal between two internal instances in this component can be mapped to physical signal(s) by a reference from the **model/view**. **interface** indicates an IP-XACT interface can be mapped from a lower-level component on this hierarchical component.

c) **drivable** (optional), when ***True***, indicates the whitebox describes a point within the IP that can be driven, i.e., forced to a new value. If ***False***, (the default), the whitebox references a point that cannot be driven. The **drivable** element is of type ***Boolean***.

d) **registerRef** (optional) names the register indicated by this whitebox when the **whiteboxType** is **register**. The **registerRef** is the full hierarchical path from the component's top-level memory map to the register, using / as a hierarchy separator. The **registerRef** element is of type ***string***.

e) **parameters** (optional) specifies any parameter names and types for a whitebox that can be parametrized.

f) **vendorExtensions** (optional) provides a space for any vendor-specific extensions.

7.15.3 Example

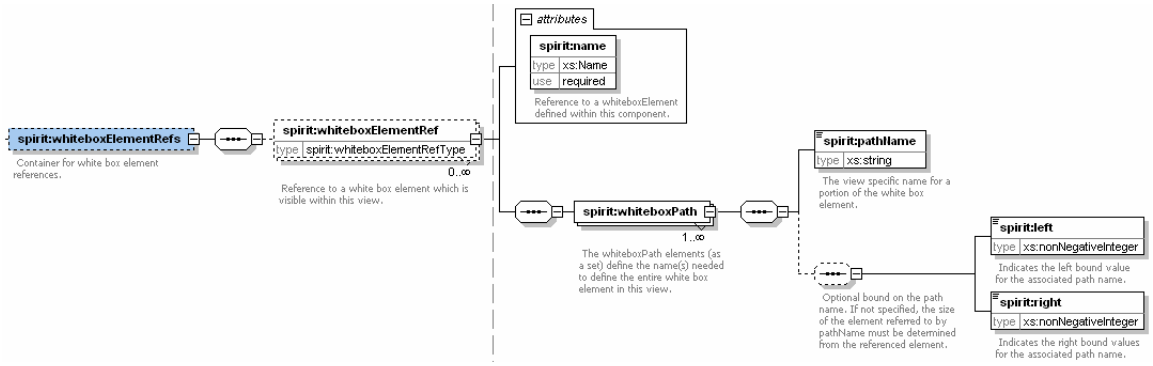
The following example shows the definition of a status register that can be accessed within a component during verification.

```
<spirit:whiteboxElements>
    <spirit:whiteboxElement>
        <spirit:name>Status</spirit:name>
        <spirit:whiteboxType>register</spirit:whiteboxType>
        <spirit:driveable>false</spirit:driveable>
        <spirit:registerRef>stat</spirit:registerRef>
    </spirit:whiteboxElement>
</spirit:whiteboxElements>
```

7.16 Whitebox element reference

7.16.1 Schema

The following schema details the information contained in the **whiteboxElementRefs** element, which may appear as an element inside the **component/model/views/view** element.



7.16.2 Description

The **whiteboxElementRefs** element contains a list of one or more **whiteboxElementRef** elements. The **whiteboxElementRef** makes a reference to a **whiteboxElement** of the component and defines the view specific path to the element. **name** (mandatory) attribute identifies the **whiteboxElement** in the containing component for which the following **whiteboxPath** applies. The **name** element is of type ***Name***. **whiteboxElement** element contains the following elements.

whiteboxPath (mandatory, unbounded) contains elements to define the path in this view to the above referenced **whiteboxElement**.

1) **pathName** (mandatory) is the language and view specific path to the location of the **whitebox-Element**. The **pathName** is of type ***string***.
2) **left** (optional, paired with **right**) sets the element bounds of the **pathName** if required by the language. The **left** element is of type ***nonNegativeInteger***.
3) **right** (optional, paired with **left**) sets the element bounds of the **pathName** if required by the language. The **right** element is of type ***nonNegativeInteger***.

7.16.3 Example

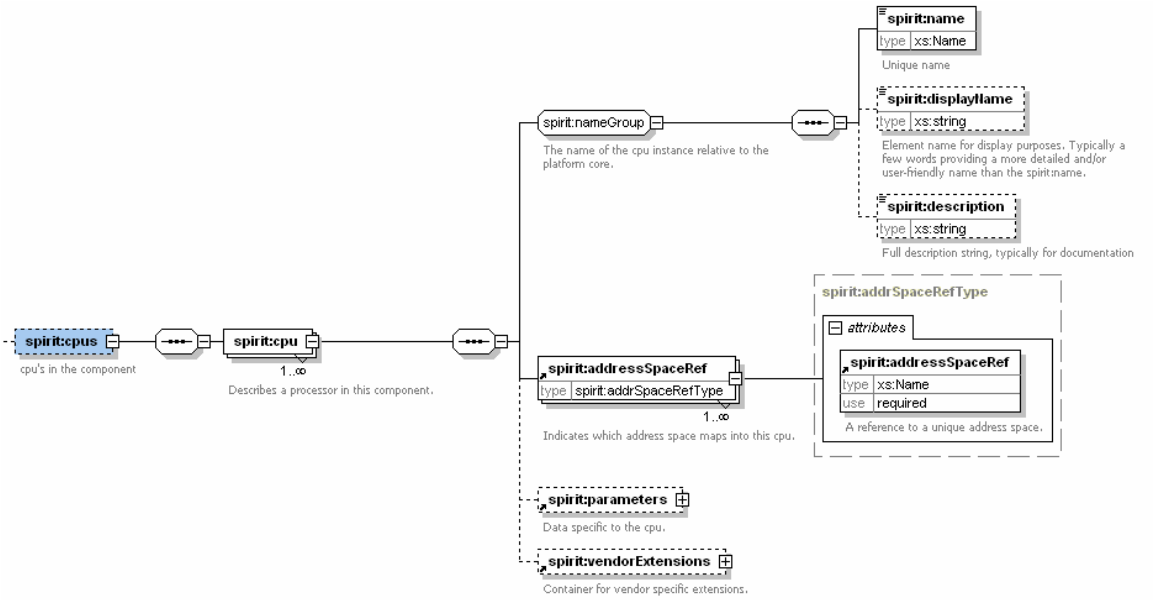
The following example shows the definition of a the whitebox path to the status register bits in a component.

```
<spirit:whiteboxElementRefs>
    <spirit:whiteboxElementRef spirit:name="Status">
        <spirit:whiteboxPath>ucontrol/ureg/status</spirit:whiteboxPath>
        <spirit:left>7</spirit:left>
        <spirit:right>0</spirit:right>
    </spirit:whiteboxElementRef>
</spirit:whiteboxElementRefs>
```

7.17 CPUs

7.17.1 Schema

The following schema details the information contained in the **CPUs** element, which may appear as an element inside the top-level **component** element.



7.17.2 Description

The **cpus** element contains an unbounded list of **cpu** elements for the containing component. The **cpu** element describes a containing component with a programmable core that has some sized address space. That same address space may also be referenced by a master interface and used to create a link for the programmable core to know from which interface transaction the software will depart.

a) ***nameGroup*** group includes the following. See X.Y.Z.
   1) **name** (mandatory) identifies the component generator. The **name** element is of type ***Name***.
   2) **displayName** (optional) allows a short descriptive text to be associated with the component generator. The **displayName** element is of type ***string***.
   3) **description** (optional) allows a textual description of the component generator. The **description** element is of type ***string***.

b) **addressSpaceRef** (required, unbounded) contains an attribute to describe information about the range of addresses with which the master interface related to this **cpu** can generate transactions.

   addressSpaceRef (mandatory) attribute references a name of an address space defined in the same component. The address space will define the range and width for transaction on this interface. See 7.7.1.

c) **parameters** (optional) specifies any **cpu**-type parameters. See X.Y.Z.

d) **vendorExtensions** (optional) adds any extra vendor-specific data related to the **cpu**.

7.17.3 Example

This example shows a simple cpu with a single **addressMap** reference.

```
<spirit:cpus>
    <spirit:cpu>
      <spirit:name>processor</spirit:name>
      <spirit:addressSpaceRef spirit:addressSpaceRef="main"/>
    </spirit:cpu>
  </spirit:cpus>
```

8. Designs descriptions

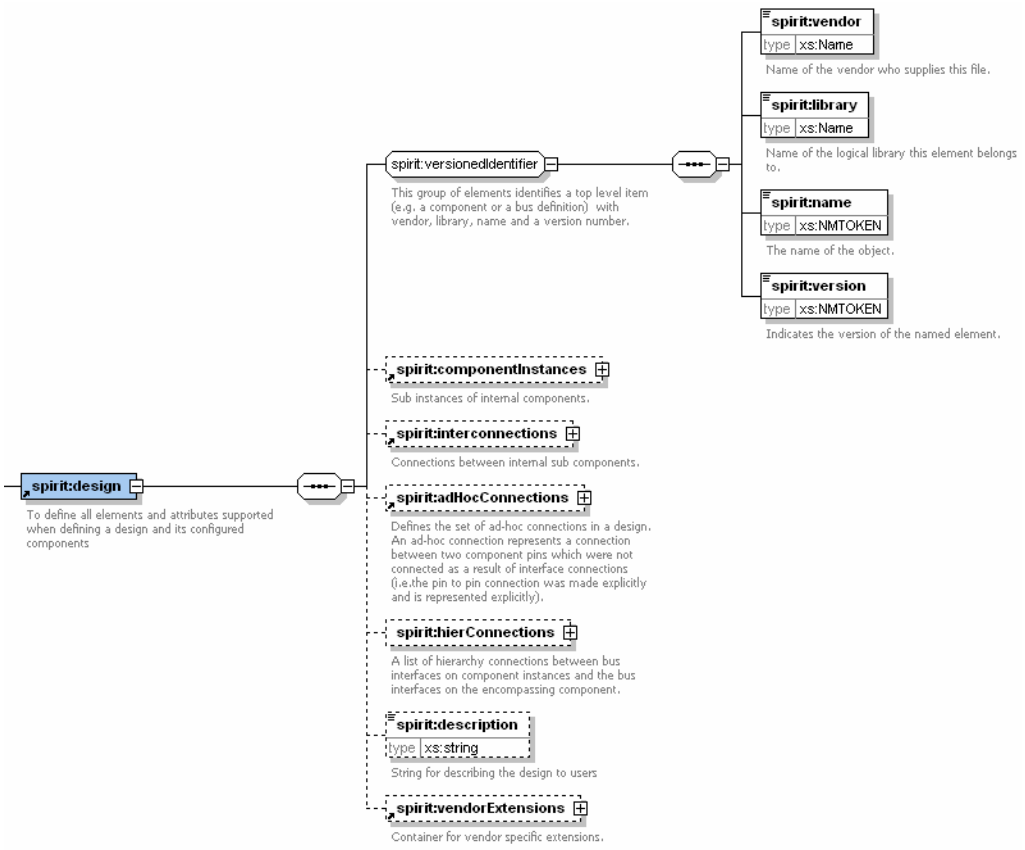
8.1 Designs

An IP-XACT *design* is the central placeholder for the collection of the assymbly of component objects meta-data. A design describes the a list of components referenced by this description, their configuration and their interconnections to each other. The interconnections may be between interfaces or between ports on a component. A design file is anologus to a schematic of components.

While a design description, with referenced components and interconnection, describes most of the information for a design, some information is missing. Such as the exact port names used by a bus interface. To resolve this a component description (refered to as a hierachical component), which contains this missing information, may contain a view with a reference to the design description to form a complete single level hierarchical description. From this point it is simple to create hierarchical descriptions by including hierachical component description in design descriptions.

8.1.1 Schema

The following schema details the information contained in the **design** element, which is one of the seven top-level elements of the schema.



8.1.2 Description

The **design** element describes the a list of referenced components, their configuration and interconnections to each other. Each element of a **design** is detailed in the rest of this clause; the main sections of a **design** are:

a) ***versionedIdentifier*** group provides a unique identifier, made up of 4 subelements for a top level IP-XACT element. See X.Y.Z for more details.

   1) **vendor** (mandatory) identifies the owner of this description. The reccomended format of the **vendor** element is the company internet domain name.

   2) **library** (mandatory) identifies a library of this description. This allows one vendor to group descriptions.

   3) **name** (mandatory) identifies a name of this description.

   4) **version** (mandatory) identifies a version of this description. This allows one vendor to provide many descriptions which all have the same name but are still uniquely identified.

b) **componentInstances** (optional) contains the list of components that are instantiated (referenced) inside the design (see 8.2).

c) **interconnections** (optional) contains the list of connections between bus interfaces of components listed inside the design(see 8.3).

d) **adHocConnections** (optional) contains a list of connections between component ports listed inside this design (see 8.5).

e) **hierConnections** (optional) contains a list of connections between a component instance's bus interface and a bus interface inside the encompassing component (see 8.6). See section on component view to see how the encompassing component can refer a design.

   This element only allows making hierarchical reference between bus interfaces. Hierarchical reference between ports is made inside the **adHocConnections** element.

f) **description** (optional) allows a textual description of the design., the **description** element is of type ***string***.

g) **vendorExtensions** (optional) adds any extra vendor-specific data related to the design.

8.1.3 Example

The following example shows as sample design with 3 components.

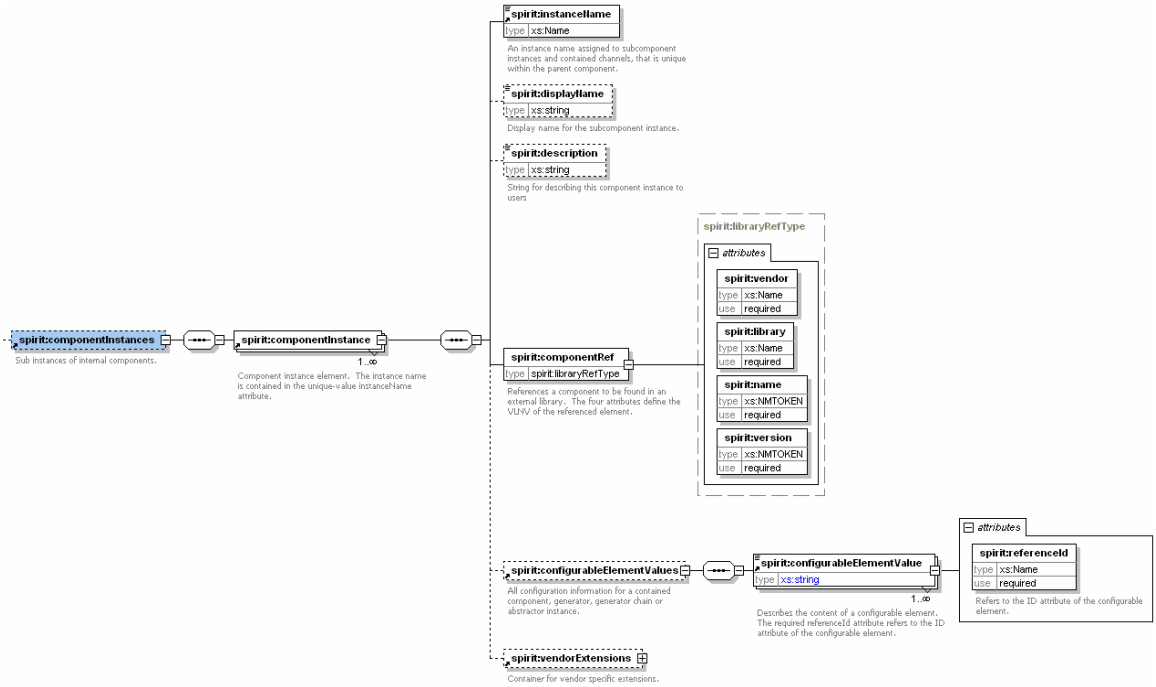
```
<spirit:design xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/
   SPIRIT/1.4" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4
   http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4/index.xsd">
   <spirit:vendor>spiritconsortium.org</spirit:vendor>
   <spirit:name>design_MCS</spirit:name>
   <spirit:version>1.0</spirit:version>
   <spirit:componentInstances>
      <spirit:componentInstance>
         <spirit:instanceName>i_ahbMaster</spirit:instanceName>
         <spirit:componentRef spirit:vendor="spiritconsortium.org"
   spirit:library="Addressing" spirit:name="ahbMaster" spirit:version="1.0"/
   >
         <spirit:configurableElementValues>
            <spirit:configurableElementValue
   spirit:referenceId="asBase">0</spirit:configurableElementValue>
            </spirit:configurableElementValues>
         </spirit:componentInstance>
      <spirit:componentInstance>
```

```
            <spirit:instanceName>i_ahbChannel12</spirit:instanceName>
            <spirit:componentRef spirit:vendor="spiritconsortium.org"
        spirit:library="Addressing" spirit:name="ahbChannel12"
        spirit:version="1.0"/>

        </spirit:componentInstance>
        <spirit:componentInstance>
            <spirit:instanceName>i_ahbSlave</spirit:instanceName>
            <spirit:componentRef spirit:vendor="spiritconsortium.org"
        spirit:library="Addressing" spirit:name="ahbSlave" spirit:version="1.0"/>
        </spirit:componentInstance>
    </spirit:componentInstances>
    <spirit:interconnections>
        <spirit:interconnection>
            <spirit:name>m2c</spirit:name>
            <spirit:activeInterface spirit:componentRef="i_ahbMaster"
        spirit:busRef="AHBMaster"/>
            <spirit:activeInterface spirit:componentRef="i_ahbChannel12"
        spirit:busRef="MirroredMaster0"/>
        </spirit:interconnection>
        <spirit:interconnection>
            <spirit:name>c2s</spirit:name>
            <spirit:activeInterface spirit:componentRef="i_ahbSlave"
        spirit:busRef="AHBSlave"/>
            <spirit:activeInterface spirit:componentRef="i_ahbChannel12"
        spirit:busRef="MirroredSlave0"/>
        </spirit:interconnection>
    </spirit:interconnections>
    <spirit:description>Addressing example, master-channel-slave</
    spirit:description>
</spirit:design>
```

8.2 Design component instances

8.2.1 Schema

The following schema details the information contained in the **componentInstances** element, which may appear as an element inside the top-level **design** element.



8.2.2 Description

The **componentInstances** element contains an unbounded list of component instances that are described inside the **componentInstance** element. This element contains the following subelements.

a) **instanceName** (mandatroy) assigns a unique name for this instance of the component in this design. The value of this element shall be unique inside a **design** element. The **instanceName** element is of type ***Name***.

b) **displayName** (optional) allows a short descriptive text to be associated with the instance. The **displayName** is of type ***string***.

c) **description** (optional) allows a textual description of the instance. The **displayName** is of type ***string***.

d) **componentRef** (mandatory) is a reference to a component description for this component instance. The componentRef element is of type ***libraryRefType*** (see X.Y.Z), it contains four attributes to specify a unique VLNV.

1) **vendor** attribute (mandatory) identifies the owner of the referenced description.

2) **library** attribute (mandatory) identifies a library of referenced description.

3) **name** attribute (mandatory) identifies a name of referenced description.

4) **version** attribute (mandatory) identifies a version of referenced description.

e) **configurableElementValues** (optional) specifies the configuration for a specific component instance by providing the value of a specific component parameter. The **configurableElementsValues** is an unbounded list of **configurableElementsValue**.

1) **configurableElementValue** (required) specifies the value to apply to a parameter, in this instance, pointed to by the **referenceId** attribute. The **configurableElementValue** is of type ***string***. The contained **referenceId** (required) is a reference to the **id** attribute of an element in the component instance. The **referenceId** attribute is of type ***Name***.

f) **vendorExtensions** (optional) adds any extra vendor-specific data related to the design.

See also: SCR 1.11.

8.2.3 Example

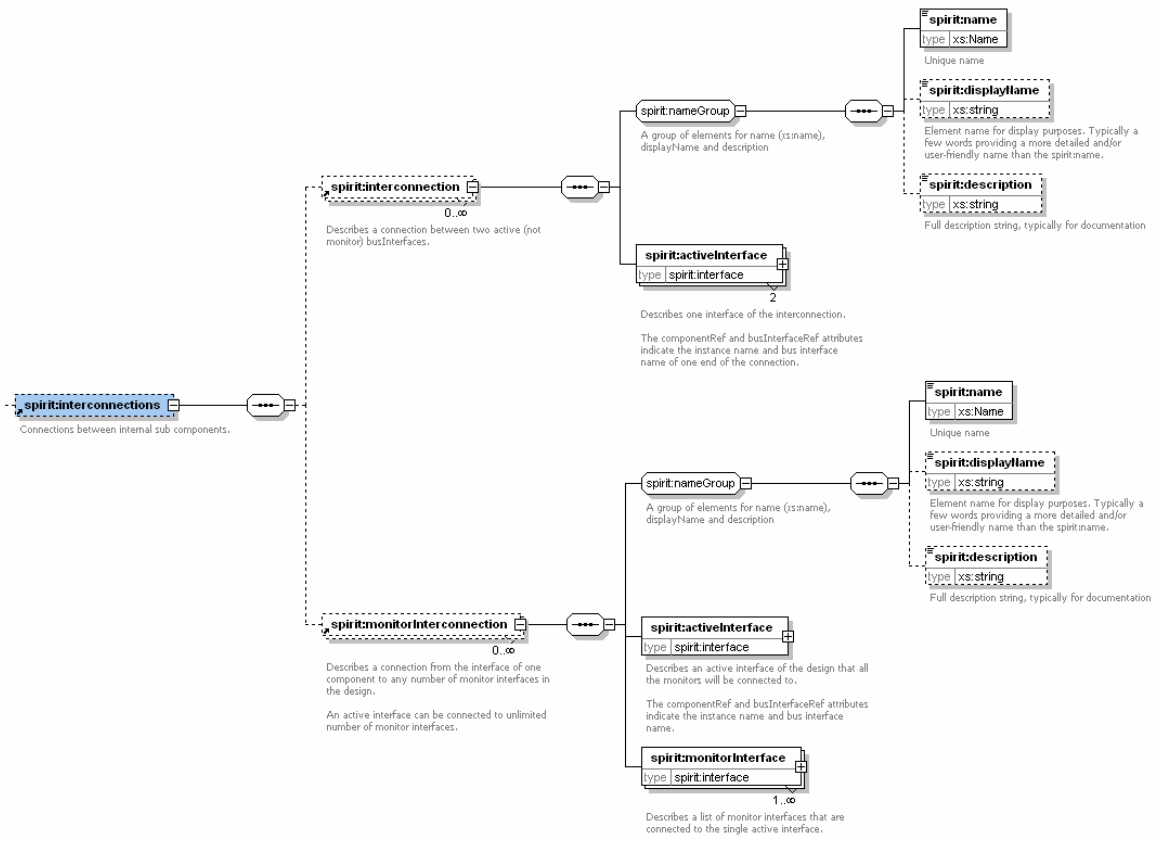
The following example shows two component instances of a design. The first one, `i_timers`, has a configurable element attach to it while the second one, `i_irqctrl`, was not configurable.

```
<spirit:componentInstances>
    <spirit:componentInstance>
        <spirit:instanceName>i_timers</spirit:instanceName>
        <spirit:componentRef spirit:vendor="spiritconsortium.org"
            spirit:library="Leon2" spirit:name="timers"
spirit:version="1.4"/>
        <spirit:configurableElementValues>
            <spirit:configurableElementValue spirit:referenceId="TPRESC">22
            </spirit:configurableElement>
        </spirit:configurableElementValues>
    </spirit:componentInstance>
    <spirit:componentInstance>
        <spirit:instanceName>i_irqctrl</spirit:instanceName>
        <spirit:componentRef spirit:vendor="spiritconsortium.org"
            spirit:library="Leon2" spirit:name="irqctrl"
spirit:version="1.4"/>
    </spirit:componentInstance>
</spirit:componentInstances>
```

8.3 Design interconnections

8.3.1 Schema

The following schema details the information contained in the **interconnections** element, which may appear as an element inside the top-level **design** element.



8.3.2 Description

The **interconnections** element contains an unbounded list of **interconnection** and **monitorInterconnection** elements. For further description on interface connections see interface connections, X.Y.Z.

a) **interconnection** (optional, unbounded) specifies a connection between one bus interface of a component and another bus interface of a component. Each interconnection contain the following elements.

1) ***nameGroup*** group includes the following. See X.Y.Z .

i) **name** (mandatory) identifies a unique name for the interconnection.

ii) **displayName** (optional) allows a short descriptive text to be associated with the connection.

iii) **description** (mandatory, 2 elements) allows a textual description of the connection.

4) **activeInterface** (optional) element specifies the two bus interfaces that are part of the interconnection. Only connections between two bus interfaces are allowed; broadcasting of interconnections is not allowed. The **activeInterface** element is of type ***interface***, see X.Y.Z.

b) **monitorInterconnections** specifies the connection between an **activeInterface** on a component and a list of **monitorInterface**s that are part of design component instances.

1) **activeInterface** (mandatroy) specifies the component bus interface to monitor; only one interface is allowed. The list of **monitorInterface**s specifies the component monitor interfaces connected to the single active interface. The **activeInterface** element is of type ***interface***, see X.Y.Z.

2) **monitorInterface** (mandatory, unbounded) specifies the connection between an **activeInterface** on a component and a list of **monitorInterface**s that are part of design component instances. There may be one or more **monitorInterconnections** specified. The **monitorInterface** element is of type ***interface***, see X.Y.Z.

See also: SCR 2.2, SCR 2.3, SCR 2.4, SCR 2.5, SCR 2.6, SCR 2.7, SCR 2.8, SCR 2.9, SCR 2.10, SCR 2.11, SCR 2.12, SCR 2.13, SCR 2.14, SCR 4.1, SCR 4.2, SCR 4.3, SCR 4.4, SCR 4.5, SCR 4.6, SCR 6.15, and SCR 6.16.

8.3.3 Example

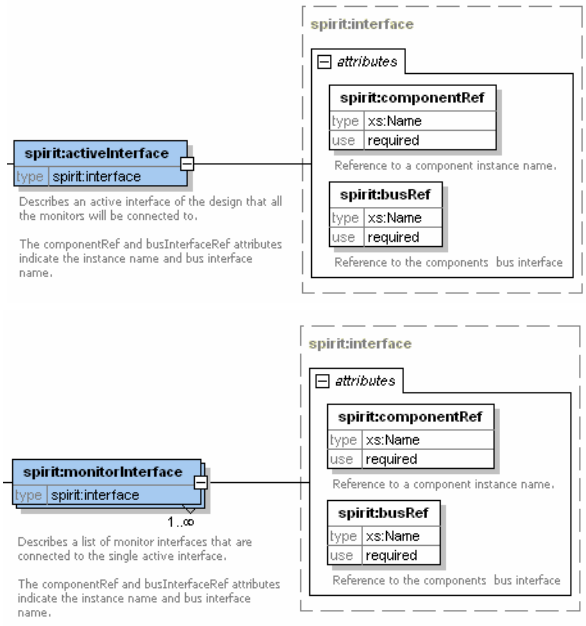
The following example shows two interconnections between three components: the interconnection `interco1` connects the interface `ambaAPB` on `i_timers` to the interface `MirroredSlave0` on `i_apbbus` while `interco2` connects the interface `ambaAPB` on `i_irqctrl` to the interface `MirroredSlave1` on `i_apbbus`.

```
<spirit:interconnections>
    <spirit:interconnection>
        <spirit:name>interco1</spirit:name>
        <spirit:activeInterface spirit:componentRef="i_timers"
                        spirit:busRef="ambaAPB"/>
        <spirit:activeInterface spirit:componentRef="i_apbbus"
spirit:busRef="MirroredSlave0"/>
    </spirit:interconnection>
    <spirit:interconnection>
        <spirit:name>interco2</spirit:name>
        <spirit:activeInterface spirit:componentRef="i_irqctrl"
spirit:busRef="ambaAPB"/>
        <spirit:activeInterface spirit:componentRef="i_apbbus"
spirit:busRef="MirroredSlave1"/>
    </spirit:interconnection>
</spirit:interconnections>
```

8.4 Design interconnection and monitor interconnection active interface

8.4.1 Schema

The following schema details the information contained in the **activeInterface** element, which may appear as an element inside the interconnection or **monitorInterconnection** element within **interconnections**.



8.4.2 Description

The **activeInterface** or **monitorInterface** element specifies the bus interface of a design component instance that is part of an interconnection or a monitor interconnection. They both have the following attributes.

a) **componentRef** (mandatory) references the instance name of a component present in the design. This component instance name needs to exist in the design.

b) **busRef** (mandatory) references one of the component bus interfaces. This specific bus interface needs to exist on the specified component instance.

8.4.3 Example

The following example shows an active interface referring the `ambaAPB` bus interface on the component instance `i_timers` and a monitor.

```
<spirit:activeInterface spirit:componentRef="i_timers"
   spirit:busRef="ambaAPB"/>
<spirit:monitorInterface spirit:componentRef="i_monitor"
   spirit:busRef="ambaAPBMonitor"/>
```

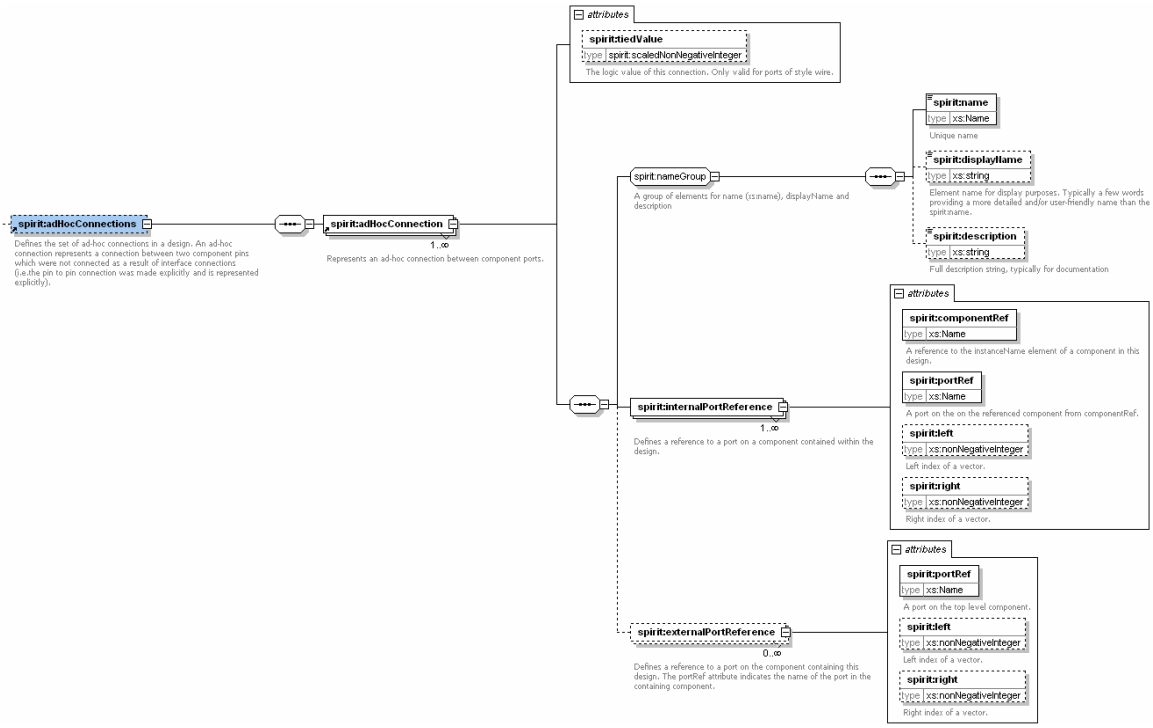
8.5 Design ad-hoc connections

The name *ad-hoc* is used for connections that are made on a port-by-port basis and not done through the higher-level bus interface. The same **ports** which make up a **busInterface** can be used in ad-hoc connections.

IP-XACT supports two cases of ad-hoc connections: the wire connection (between ports having a wire style) and the transactional connection (between ports having a transactional style). The direct connection between a wire-style port and a transactional-style port is not allowed; a specific adapter component needs to be inserted in between them.

8.5.1 Schema

The following schema details the information contained in the **adHocConnections** element, which may appear as an element inside the top-level **design** element.



8.5.2 Description

The **adHocConnections** element contains an unbounded list of **adHocConnection** elements. An **adHocConnection** specifies connections between component instance ports or between component instance ports and ports of the encompassing component (in the case of a hierarchical component). Each **adHocConnection** element has a **tiedValue** (optional) attribute that specifies a fixed logic (1 and 0) value for this connection. The **tiedValue** attribute is of type ***scaledNonNegativeInteger***. The **adHocConnection** element contains the following subelements.

a) ***nameGroup*** group includes the following. See X.Y.Z.

   1) **name** (mandatory) identifies a unique name for the interconnection.

   2) **displayName** (optional) allows a short descriptive text to be associated with the connection.

   3) **description** (mandatory, 2 elements) allows a textual description of the connection.

b) **internalPortReference** (mandatory, unbounded) references the port of a component instance. This element has four attributes.

   1) **componentRef** (mandatory) references the component instance name for the port. The **componentRef** attribute is of type ***Name***.

   2) **portRef** (mandatory) references the port name on the specific component instance. The **portRef** attribute is of type ***Name***.

   3) **left** and **right** (optional) specify a portion of the port range. The **left** and **right** attribute is of type ***nonNegativeInteger***.

c) **externalPortReference** (optional, unbounded) references a port of the encompassing component where this design is referred (for hierarchical ad-hoc connections). This element has three attributes.

1) **portRef** (mandatory) references the port name on the encompassing component. The **portRef** attribute is of type ***Name***.

2) **left** and **right** (optional) specify a portion of the port range. The **left** and **right** attribute is of type ***nonNegativeInteger***.

See also: SCR 6.15 and SCR 6.16.

8.5.3 Example

The following example shows two ad-hoc connections. The first one, `d1e1074`, is done between port `irlin` on component instance `i_irqctrl` and port `irqvec` on component instance `i_leon2Proc`. The second one, `i_leon2Proc_mresult`, is made between port `mresult` on component instance `i_leon2Proc` and port `i_leon2Proc_mresult` of the encompassing component.

```
<spirit:adHocConnections>
    <spirit:adHocConnection>
        <spirit:name>d1e1074</spirit:name>
        <spirit:internalPortReference spirit:componentRef="i_irqctrl"
spirit:portRef="irlin" spirit:left="3"
            spirit:right="0"/>
        <spirit:internalPortReference spirit:componentRef="i_leon2Proc"
spirit:portRef="irqvec"
            spirit:left="3" spirit:right="0"/>
    </spirit:adHocConnection>
    <spirit:adHocConnection>
        <spirit:name>i_leon2Proc_mresult</spirit:name>
        <spirit:internalPortReference spirit:componentRef="i_leon2Proc"
spirit:portRef="mresult"
            spirit:left="31" spirit:right="0"/>
        <spirit:externalPortReference spirit:portRef="i_leon2Proc_mresult"/
>
    </spirit:adHocConnection>
</spirit:adHocConnections>
```

8.5.4 Ad-hoc wire connection

For ad-hoc connections between wire-style ports, IP-XACT requires:

— The style of each port be the same style (i.e., **wire**).

— The **direction**s match as described in Table 8.

— The sizes of each port `(max(left,right) - min(left,right) + 1)` are exactly the same and their bits are connected from left-to-right with no exceptions. In the **internalPortReference** element, **left** and **right** only define the size of the portion of the port that is connected.

Table 8—Direction requirements

Direction	in	out	inout
in	yes	yes	yes
out	yes	no	yes
inout	yes	yes	yes

Example

This is an example of these rules being applied.

```
<spirit:adHocConnection>
   </spirit:internalPortReference componentRef="U1" portRef="A"
      left="8" right="1">
   </spirit:internalPortReferencenal componentRef="U2" portRef="B"
      left="7" right="0">
</spirit:adHocConnection>
```

Implies these connections:

```
U1/A[8] = U2/B[7]
U1/A[7] = U2/B[6]
U1/A[6] = U2/B[5]
U1/A[5] = U2/B[4]
U1/A[4] = U2/B[3]
U1/A[3] = U2/B[2]
U1/A[2] = U2/B[1]
U1/A[1] = U2/B[0]
```

NOTE—The **typeName**s do not have to match between the two ports, it is up to the DE or simulator to potentially resolve unmatching types, e.g., it is possible to connect a VHDL `std_logic` port to a SystemC `sc_in` *`bool`* port.

8.5.5 Ad-hoc transactional connection

For ad-hoc transactional connections, IP-XACT requires:

— The style of each port be the same style (i.e., **transactional**).

— The **transTypeDef**/**typeName** name of each port are the same (e.g., `sc_port`).

— The **initiative**s match as described in Table 9.

Table 9—Initiative requirements

Initiative	requires	provides	both
requires	yes	yes	yes
provides	yes	no	yes
both	yes	yes	yes

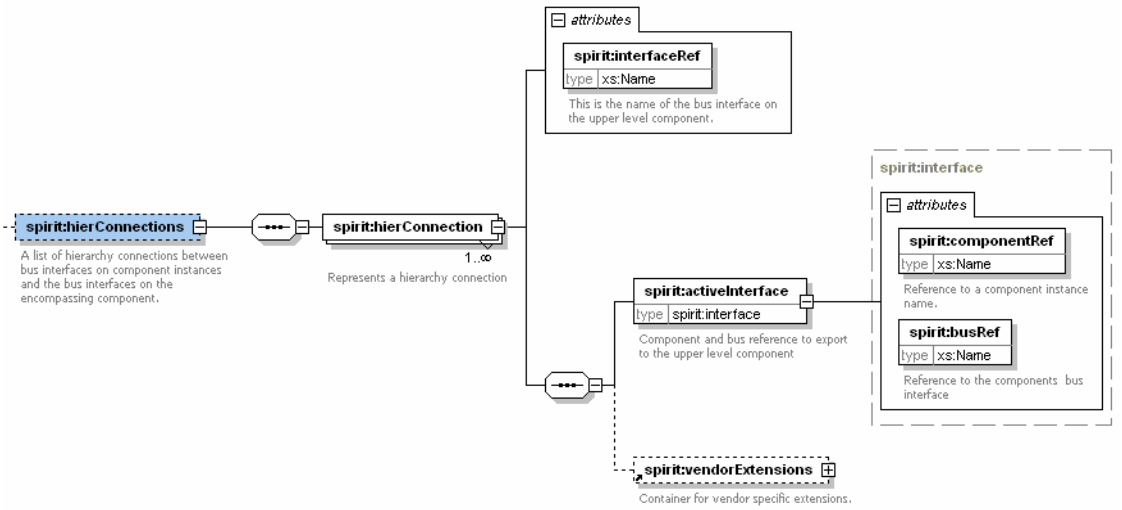
— The **service**/**serviceTypeDef**/**typeName**s match.

Furthermore, two ports with a **requires** initiative can be connected. This means they would both connect to a mediated link (e.g., a wire, buffer, FIFO, or any complex link) in a top SystemC or SystemVerilog netlist. This mediated link provides the protocol interfaces required by each port. The name, type, and parameters of this mediated link are not defined by IP-XACT, but could be given as input to a netlister generator.

8.6 Design hierarchical connections

8.6.1 Schema

The following schema details the information contained in the **hierConnections** element, which may appear as an element inside the top-level **design** element.



8.6.2 Description

The **hierConnections** element contains an unbounded list of **hierConnection** elements. **hierConnection** represents a hierarchical interface connection between a bus interface on the encompassing component and a bus interface on a component instance of the design. **hierConnection** contains an **interfaceRef** (mandatory) attribute that provides one end of the interconnection; it is the name of the bus interface on the encompassing component. The **interfaceRef** attribute is of type ***Name***. The **hierConnection** element contains the following elements and attributes.

a) **activeInterface** (mandatroy) specifies the component instance bus interface for connection to the encompassing component, only one **activeInterface** is allowed. The **activeInterface** element is of type ***interface***, see X.Y.Z.

b) **vendorExtensions** (optional) adds any extra vendor-specific data related to the hierarchical interface connection.

See also: SCR 10.1, SCR 10.2, SCR 10.3, SCR 10.4, SCR 10.5, SCR 10.6, SCR 10.7, SCR 10.8, SCR 10.9, SCR 10.11, SCR 10.12, SCR 10.13, and SCR 10.14.

8.6.3 Example

The following example shows a hierarchical interconnection between the `AHBReset_1` bus interface on the encompassing component and the `AHBReset` bus interface on the `i_ahbbus` component instance.

```
<spirit:hierConnections>
   <spirit:hierConnection spirit:interfaceRef="AHBReset_1">
      <spirit:activeInterface spirit:componentRef="i_ahbbus"
spirit:busRef="AHBReset"/>
   </spirit:hierConnection>
</spirit:hierConnections>
```

9. Abstractor descriptions

Designs that incorporate IP models using different modeling styles (e.g., TLM and RTL modeling styles) may contain interconnections between such components using different abstractions of the same bus type. A DE may describe how such interconnections are to be made using an abstractor. Unlike a component, an abstractor is not referenced from a design file, but instead is referenced from a design configuration file. See the design configuration file section 4.4 for more information on referencing abstractors. IP-XACT can:

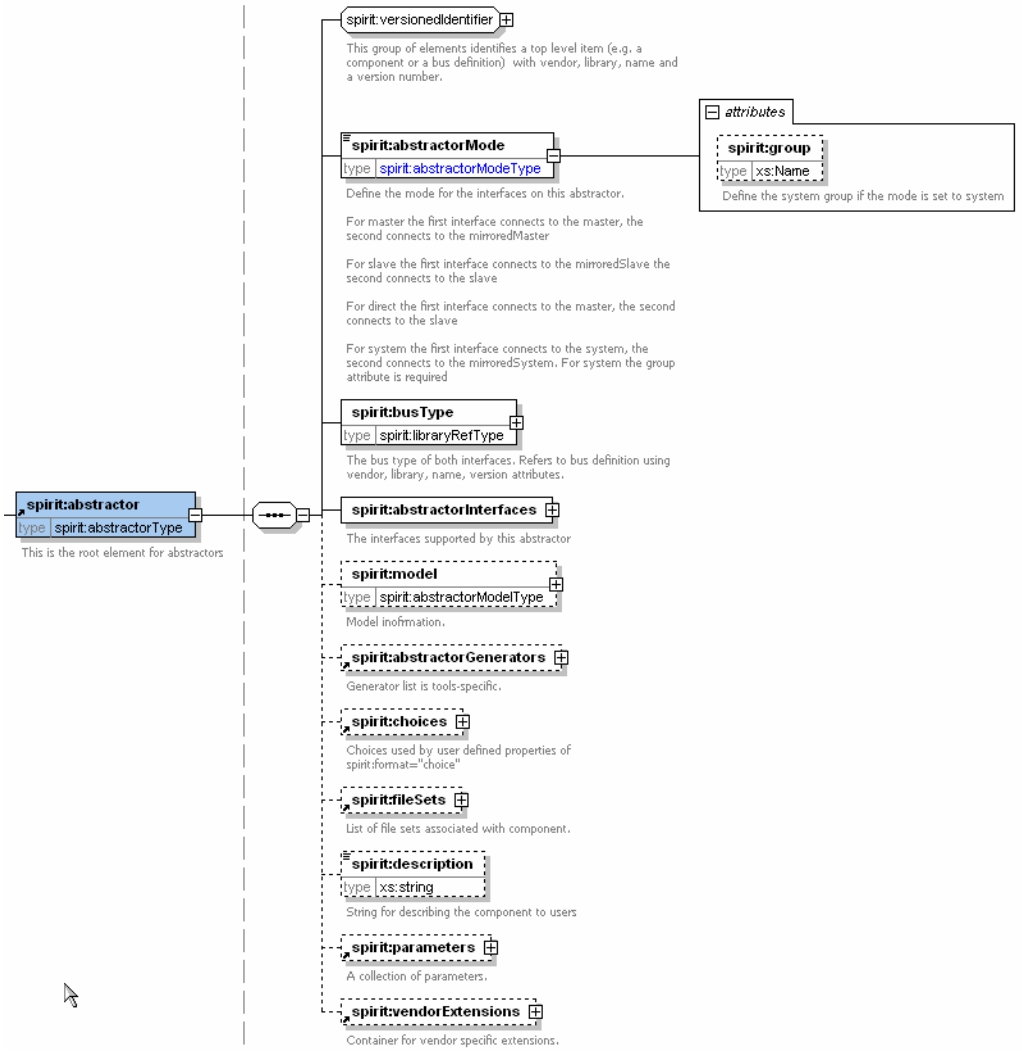
- Model different level of abstraction for the same bus type through the use of abstraction definitions.
- Model special-purpose components called “abstractors” to bridge between two different abstraction of the same bus type.
- Extend the design configuration file to allow DEs to generate designs that include these abstractors where needed.

This chapter defines abstractors and describes how to model them as IP-XACT objects.

9.1 Abstractors

9.1.1 Schema

The following schema details the information contained in the **abstractor** element, which is one of the seven top-level elements in the IP-XACT specification used to describe an abstractor.



9.1.2 Description

The **abstractor** element has two (mandatory) interfaces, called **abstractorInterfaces**. An **abstractor** also contains the following elements.

a) Mandatory elements

1) **abstractorInterfaces** are interfaces having the same bus type, but differing abstraction types (see 9.2).

2) **versionedIdentifier** is a unique VLNV identifier.

3) **abstractorMode** determines the interface mode of these interfaces.

i) **master** specifies the first interface connects to the master, the second connects to the mirrored-master.

ii) **slave** specifies the first interface connects to the mirrored-slave, the second connects to the slave.

iii) **direct** specifies the first interface connects to the master, the second connects to the slave.

iv) **system** specifies the first interface connects to the system, the second connects to the mirrored-system; in this case, the **group** attribute is also required.

3) **busType** defines the VLNV of the **busDefinition** of the two **abstractorInterfaces**.

b) Optional elements

1) **model** defines the abstractor views, its ports, and its model parameters (see 9.3).

2) **abstractorGenerators** defines any generators applying to the abstractor (see 9.6).

3) The remaining elements: **choices**, **fileSets**, **description**, **parameters**, and **vendorExtensions** are the same as those defined for the **component**. **Add xref OR copy that material here??

See also: SCR 1.13 and SCR 3.23.

9.1.3 Example

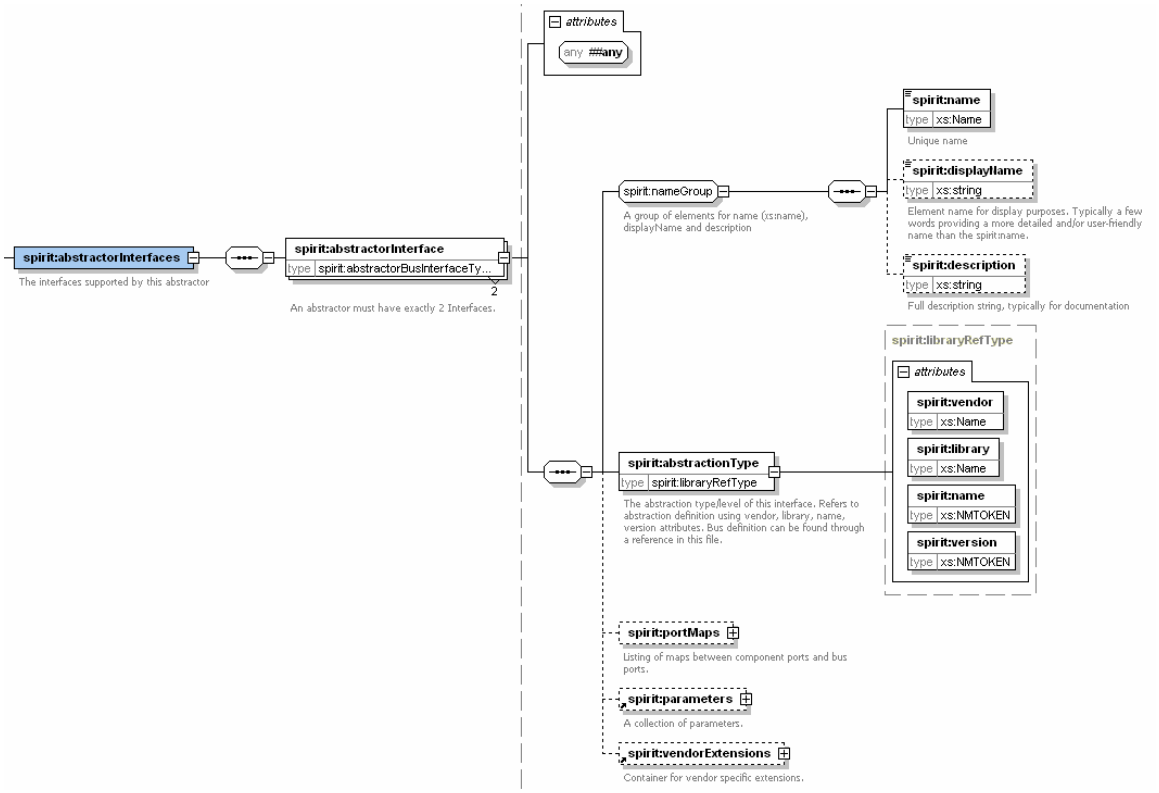
The following example shows a simple slave abstractor having AHB PV and AHB PVT interfaces.

```
<spirit:abstractor>
                    <spirit:vendor>spiritconsortium.org</spirit:vendor>
                    <spirit:library>Leon2</spirit:library>
                    <spirit:name>pv2rtl</spirit:name>
                    <spirit:version>1.4</spirit:version>
                    <spirit:abstractorMode>slave</spirit:abstractorMode>
                    <spirit:abstractorInterfaces>
                        <spirit:abstractorInterface>
                            <spirit:name>PVinterface</spirit:name>
                            <spirit:abstractionType
                                spirit:vendor="spiritconsortium.org"
                                spirit:library="Leon2"
                                spirit:name="AHB_PV"
                                spirit:version="1.0"/>
                        </spirit:abstractorInterface>
                        <spirit:abstractorInterface>
                            <spirit:name>PVTinterface</spirit:name>
                            <spirit:abstractionType
                                spirit:vendor="spiritconsortium.org"
                                spirit:library="Leon2"
                                spirit:name="AHB_PVT"
                                spirit:version="1.0"/>
                        </spirit:abstractorInterface>
                    </spirit:abstractorInterfaces>
                    <spirit:abstractorModel>
                    <spirit:abstractorGenerators>
</spirit:abstractor>
```

9.2 Abstractor interfaces

9.2.1 Schema

The following schema defines the information contained in the **abstractorInterfaces** element, which appears within an **abstractor** object.



9.2.2 Description

The **abstractorInterfaces** element defines the two abstraction interfaces of the abstractor. Each **abstractorInterface** contains the following elements.

a) Mandatory elements

1) **name** (included in **nameGroup**) identifies the abstraction interface. **nameGroup** can also have two additional (optional) subelements: **displayName**, which allows a short descriptive text to be associated with the abstraction interface, and **description** which allows a textual description of the abstraction interface.

2) **abstractionType** is the VLNV of an abstraction definition.

b) Optional elements

1) **portMap** defines the mapping between the abstractor ports and the logical ports defined in the referenced **abstractionDefinition**. This schema is the same as the **portMap** schema defined in a **component**.**Add xref OR copy that material here??

2) The elements **parameters** and **vendorExtensions** and the vendor attributes `(xs:any)` are the same as those defined for the **component**. **Add xref OR copy that material here??

9.2.3 Example

This example shows a port within an abstraction definition containing a single timing constraint. On a master interface, the port gets 40% of the cycle time and on a mirrored master interface, it gets 60% of the cycle time.

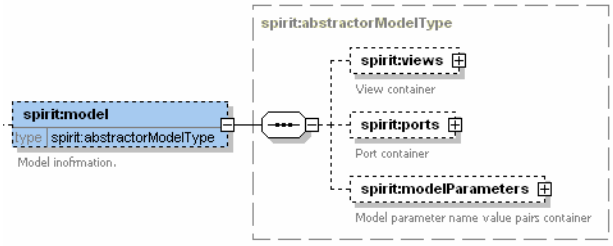
```
<spirit:port>
                        <spirit:logicalName>HRDATA</spirit:logicalName>
```

```
                    <spirit:wire>
                        <spirit:onMaster>
                            <spirit:modeConstraints>
                                <spirit:timingConstraint
    spirit:clockName="HCLK">40
                                </spirit:timingConstraint>
                            </spirit:modeConstraints>
                            <spirit:mirroredModeConstraints>
                                <spirit:timingConstraint
    spirit:clockName="HCLK">60
                                </spirit:timingConstraint>
                            </spirit:mirroredModeConstraints>
                        </spirit:onMaster>
                    </spirit:wire>
</spirit:port>
```

9.3 Abstractor models

9.3.1 Schema

The following schema defines the information contained in the abstractor **model** element, which may appear within an **abstractor** object.



9.3.2 Description

The abstractor **model** element defines the abstractor **views** (see 9.4), **ports** (see 9.5), and **modelParameters**. Each of which is described in the following sections. **Where (chapter 7); let's add this xref [for **modelParameters**]?? Is all the **ports** material covered in 9.5??

9.3.3 Example

The following example shows an abstractor model with a single SystemC view, two transactional ports, and a constructor model parameter.

```
<spirit:model>
                    <spirit:views>
                        <spirit:view>
                            <spirit:name>systemCView</spirit:name>
                            <spirit:envIdentifier>:*Simulation:</
    spirit:envIdentifier>
                            <spirit:language>systemc2.1</spirit:language>
                            <spirit:modelName>pv2pvt</spirit:modelName>
                            <spirit:fileSetRef>abstractorFileSetRef</
    spirit:fileSetRef>
                        </spirit:view>
```

```
                        </spirit:views>
                        <spirit:ports>
                            <spirit:port>
                                <spirit:name>pv_slave</spirit:name>
                                <spirit:transactional>
                                    <spirit:service>
                                        <spirit:initiative>provides</
    spirit:initiative>

    <spirit:serviceTypeDefs><spirit:serviceTypeDef>
                                            <spirit:typeName>trans_if</
    spirit:typeName>
                                        </spirit:serviceTypeDef></
    spirit:serviceTypeDefs>
                                    </spirit:service>
                                </spirit:transactional>
                            </spirit:port>
                            <spirit:port>
                                <spirit:name>pvt_master</spirit:name>
                                <spirit:transactional>
                                    <spirit:service>
                                        <spirit:initiative>requires</
    spirit:initiative>

    <spirit:serviceTypeDefs><spirit:serviceTypeDef>
                                            <spirit:typeName>req_rsp_if</
    spirit:typeName>
                                        </spirit:serviceTypeDef></
    spirit:serviceTypeDefs>
                                    </spirit:service>
                                </spirit:transactional>
                            </spirit:port>
                        </spirit:ports>
                        <spirit:modelParameters>
                            <spirit:modelParameter spirit:usageType="nontyped">
                                <spirit:name>moduleName</spirit:name>
                                <spirit:value
spirit:id="moduleNameId" spirit:resolve="user">ABSTRACTOR_PV2PVT
                                </spirit:value>
                            </spirit:modelParameter>
                        </spirit:modelParameters>
</spirit:model>
```

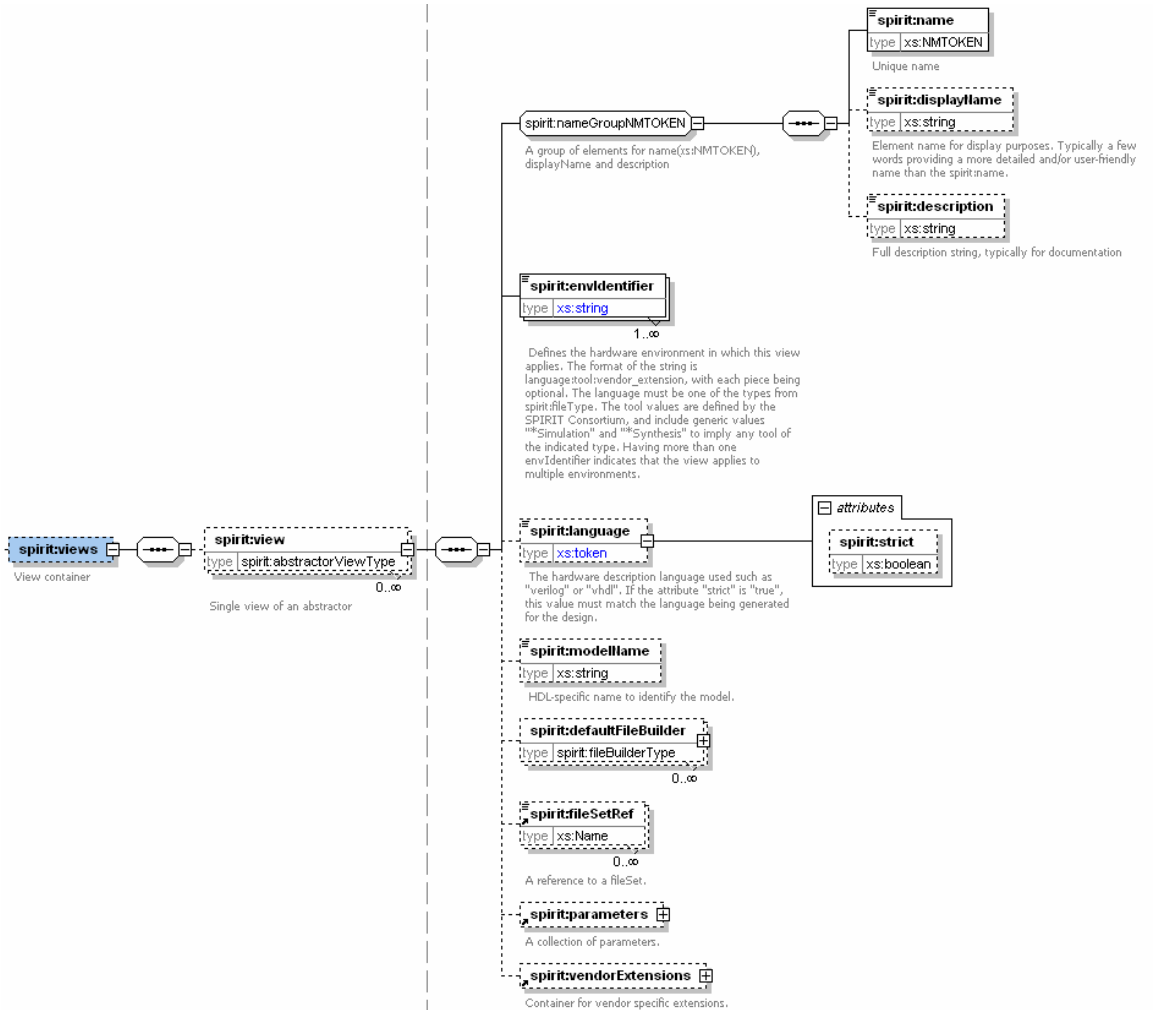
9.4 Abstractor views

9.4.1 Schema

The following schema defines the information contained in the **view** element, which appears within the **views** element of an **abstractor**.

This schema is almost identical to the **component view** (see xref), except:

- Abstractors have no **hierarchyRef** element.
- Abstractors have no **constraintSetRef** element.
- Abstractors have no **whiteboxElementRefs** element.



9.4.2 Description

The **view** element defines the different abstractor **views**. See the component view description for the definition of each element and attributes. **Add xref OR copy that material here??

The following restrictions apply to abstractor **view** elements.

a) The **envIdentifier** shall only define simulation tools.

b) The **language** needs to support a mix of the two abstraction definitions described in the abstractor (e.g., a TLM to RTL abstractor would need a language, such as SystemC, supporting both a transactional abstract level description and an RTL description).

9.4.3 Example

This example shows two abstractor views: a SystemC view and a SystemVerilog view. Such a configuration assumes the abstractor ports can be expressed with a generic **typeDef** that is supported in both languages.

```
<spirit:views>
    <spirit:view>
        <spirit:name>systemCView</spirit:name>
```

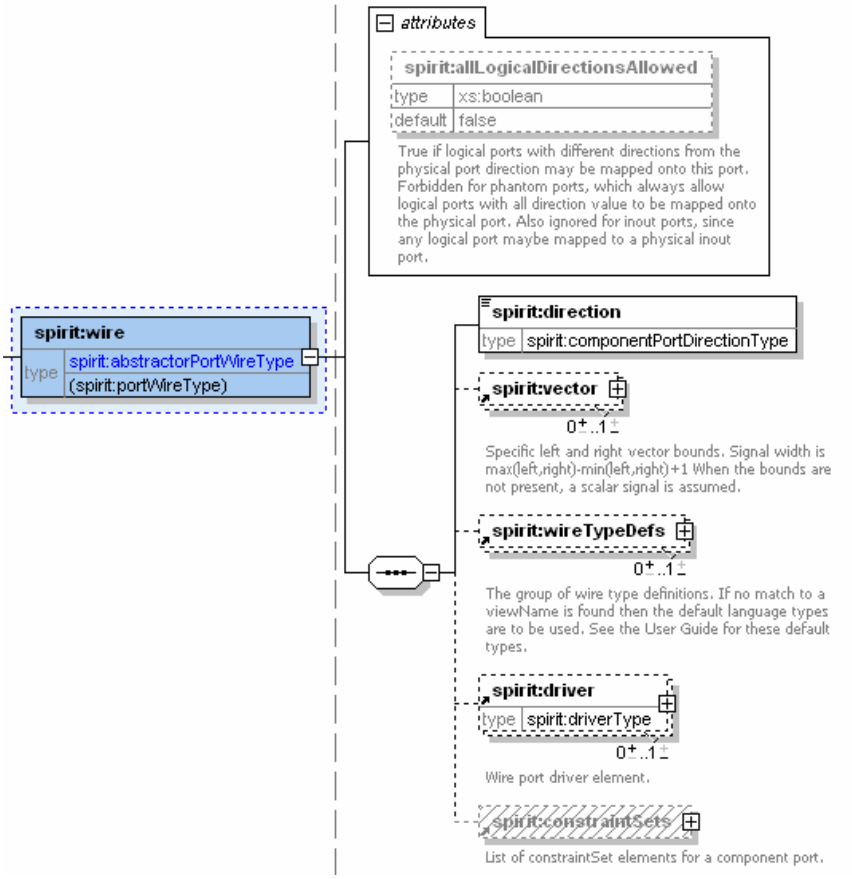
```
                        <spirit:envIdentifier>:*Simulation:</
spirit:envIdentifier>
                        <spirit:language>systemc2.1</spirit:language>
                        <spirit:modelName>pv2pvt</spirit:modelName>
                        <spirit:fileSetRef>scFileSetRef</spirit:fileSetRef>
                    </spirit:view>
                    <spirit:view>
                        <spirit:name>systemVView</spirit:name>
                        <spirit:envIdentifier>:*Simulation:</
spirit:envIdentifier>
                        <spirit:language>systemVerilog</spirit:language>
                        <spirit:modelName>pv2pvt</spirit:modelName>
                        <spirit:fileSetRef>svFileSetRef</spirit:fileSetRef>
                    </spirit:view>
                </spirit:views>
```

9.5 Abstractor ports

abstractor ports are almost identical to **component ports**; the abstractor transactional ports are exactly the same as the component transactional ports. The **abstractor wire ports** defined here only differ from **component wire ports** by the absence of the **constraintSet** element, because implementation constraints are not needed for abstractors.

9.5.1 Schema

The following schema element defines the information contained in the **wire** element, which appears within an **abstractor port**.



9.5.2 Description

The **wire** element is used to define wire ports described in the abstractor. See the component wire port description for the definition of each element and attributes.**Add xref OR copy that material here??

9.5.3 Example

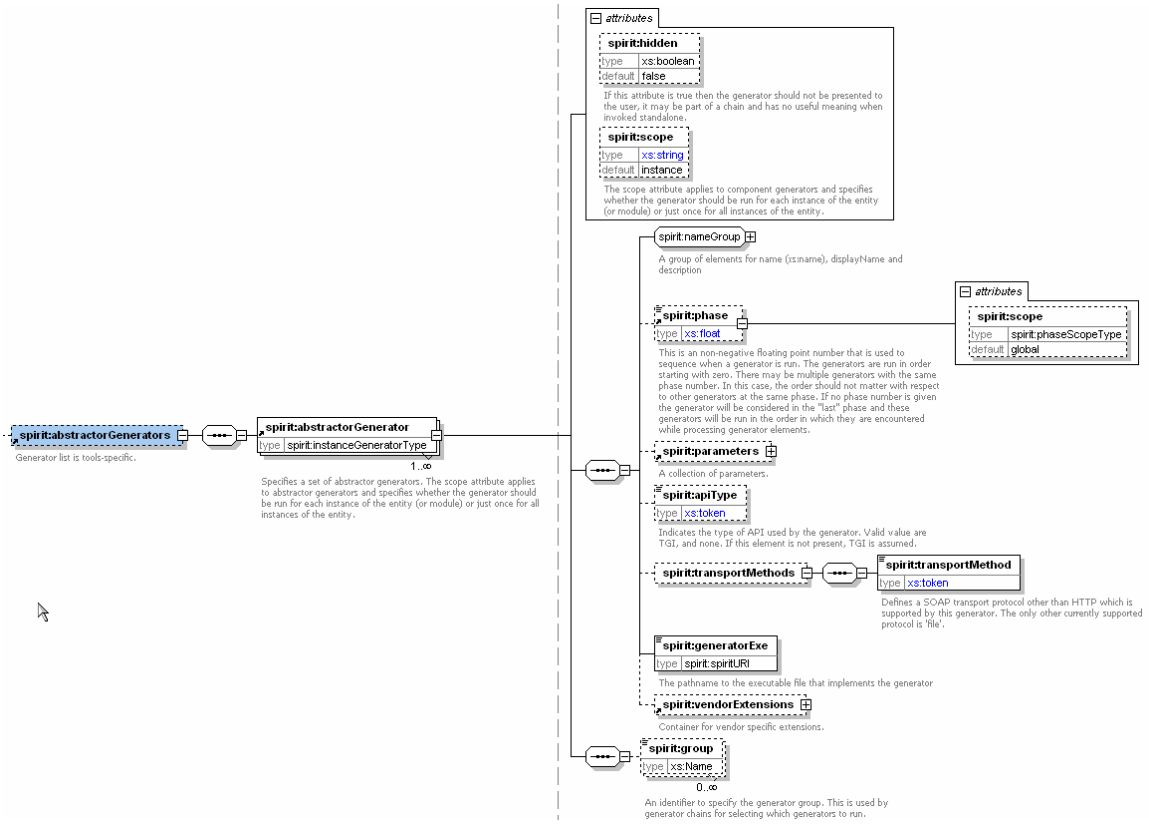
The following example shows a simple address port of 32 bits.

```
<spirit:port>
                    <spirit:name>paddr</spirit:name>
                    <spirit:wire>
                        <spirit:direction>in</spirit:direction>
                        <spirit:vector>
                            <spirit:left>31</spirit:left>
                            <spirit:right>0</spirit:right>
                        </spirit:vector>
                    </spirit:wire>
</spirit:port>
```

9.6 Abstractor generators

9.6.1 Schema

The following schema defines the information contained in the **abstractorGenerators** element, which may appear within an **abstractor** object.



9.6.2 Description

The **abstractorGenerators** element defines any generators applying to an abstractor. The **abstractor Generator** has exactly the same schema definition as a **componentGenerator**.

See the component generator description for the definition of each element and attributes. **Add xref OR copy that material here??

9.6.3 Example

The following example shows a document generator attached to an abstractor. This generator is a TCL script that can be executed as `tclsh generatorExe parameter`. In this example, the parameter is a configurable parameter named `useDefaultValues`. This generator uses the TGI API with a SOAP transport protocol based on file.

```
<spirit:abstractorGenerator>
                    <spirit:name>genAbstractorDoc</spirit:name>
                    <spirit:parameters>
                        <spirit:parameter>
```

```
                        <spirit:name>useDefaultValues</spirit:name>
                        <spirit:value spirit:id="sdvId"
    spirit:resolve="user">true
                        </spirit:value>
                     </spirit:parameter>
                  </spirit:parameters>
                  <spirit:apiType>TGI</spirit:apiType>
                  <spirit:transportMethods>
                           <spirit:transportMethod>file</
    spirit:transportMethod>
                  </spirit:transportMethods>
                  <spirit:generatorExe>../bin/absDocGen.tcl</
    spirit:generatorExe>
                  <spirit:group>genDocs</spirit:group>
</spirit:abstractorGenerator>
```

10. Generators

IP-XACT defines a tool integration schema that provides a standard method for linking applications (external generators and tool plug-ins) into a DE, enabling a more flexible, optimized development environment. IP-XACT enabled tools can interpret, configure, integrate, and manipulate IP blocks which comply with the IP meta-data description by using the IP-XACT generator interface.

This *genarator interface* allows the querying of XML IP meta-data which has been imported into the design-environment, including inquires about the existence of IP, the structure of IP, or features offered by that IP, such as configurability and interface protocol support. The generator interface can also be used by a generator to import or export meta-data when an IP block is extracted from or imported back into the DE.

This interface also serves as an interface to generators and tool plug-ins, allowing the execution of these scripts and code-elements against the SoC meta-description. Plus, it enables the registration of new generators or plug-ins, exporting SoC meta-data and updating that data following generator or plug-in execution, and handling of generator or plug-in error conditions which relate to the meta-data description.

10.1 Tight integration

In IP-XACT, a *tight integration* of an interface means the direct interfacing to generators and XML meta-data within the DE, as shown in Figure 12. A Tight Genarator Interface (TGI) can manipulate values of elements, attributes, and parameters for IP-XACT compliant XML.

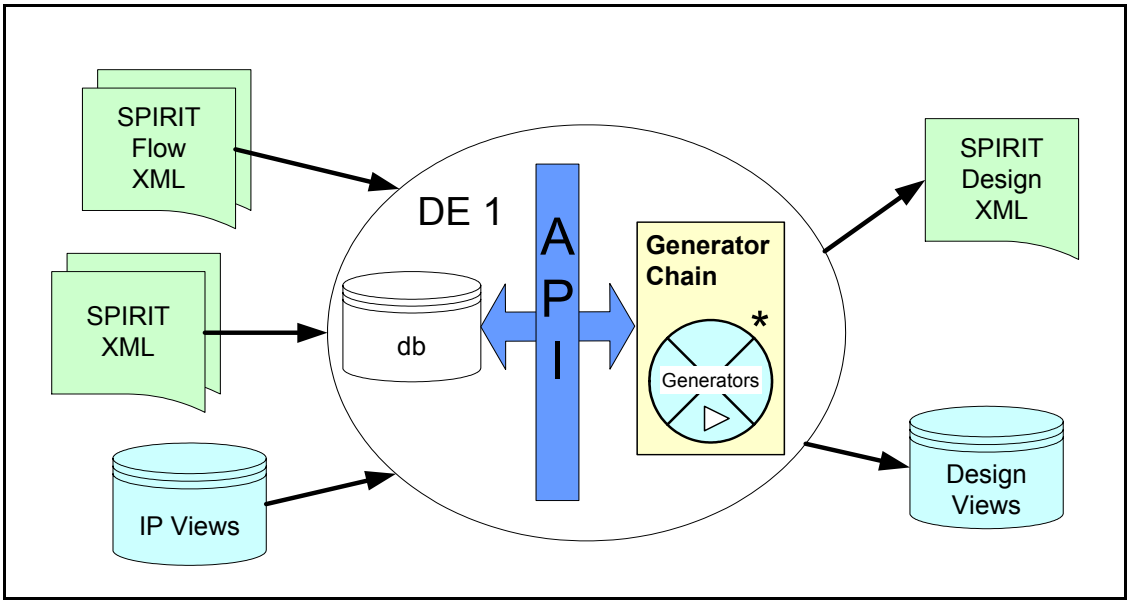


Figure 12—Example of tight integration flow

--> Replace API by TGI in the preceding figure

The DE reads the XML input files and the internal database representation is accessed via a TGI, which is a means of accessing and modifying the IP-XACT data from within an external program invoked via a generator. The results of these generators can be used to update the database until the design and all its configurable parameters are finally saved to an XML file. For more information on using the TGI, see the following document: http://www.spiritconsortium.org/releases/tgi/index.html.

For 1.4, this will be shown as an external reference to the draft Standard

10.2 Generator chain

In IP-XACT, a design flow can be represented as a generator chain that links an ordered sequence of named tasks. Each named task can be represented as a single generator or as a generator chain. This way, design flow hierarchies can be constructed and executed from within a given DE. The DE itself is responsible for understanding the semantics of the specified chain described in the XML schema.

The generator group and its elements are defined in the `generator.xsd` file. In addition:

- A *generator group* is a named generator containing a sequential list of generator invocations.
- A *generator chain* is a sequential list of ordered generator groups.
- A *generator invocation* is a method of running an application at a defined phase in the generator group using a given number of parameters.
- A *phase* is a number that defines when a generator invocation occurs in a sequential ascending order.
- The behavior of the generator invocation can also be influenced.
- While the generator group names are generic (and use string values), the names of generators should reflect what they are trying to achieve.

10.3 Phase numbers

Phase numbers are intended to define the sequence in which generators are fired. A *phase number* is a non-negative floating-point number that is used to sequence when a generator is run. A series of generators and phase number-specific sequences of named task invocations can be built to influence when a DE fires a specific generator. Generators can be attached to high-level chains or specific components.

Multiple generators can contain the same phase number, as shown in Figure 13.

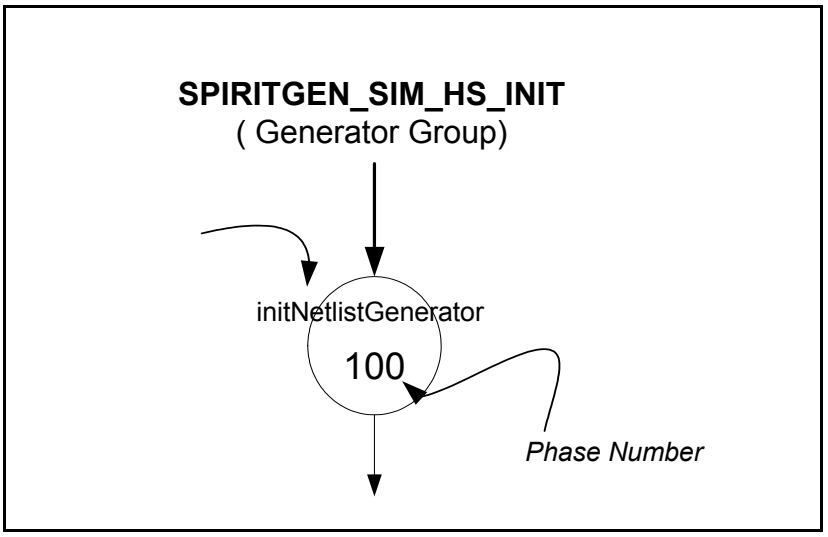


Figure 13—Generator example with phase number

In this case, the order does not matter with respect to other generators at the same phase. If no phase number is given, then the DE can decide the generator's position.

Generators can be attached to both components by using the same generator group name. In this case, the sequence for invoking each generator depends on the associated phase number. It is up to the DE to process the generator chains, groups, and phase numbers to construct the sequence of generator invocations.

The following XML file specifies a call to such a generator.

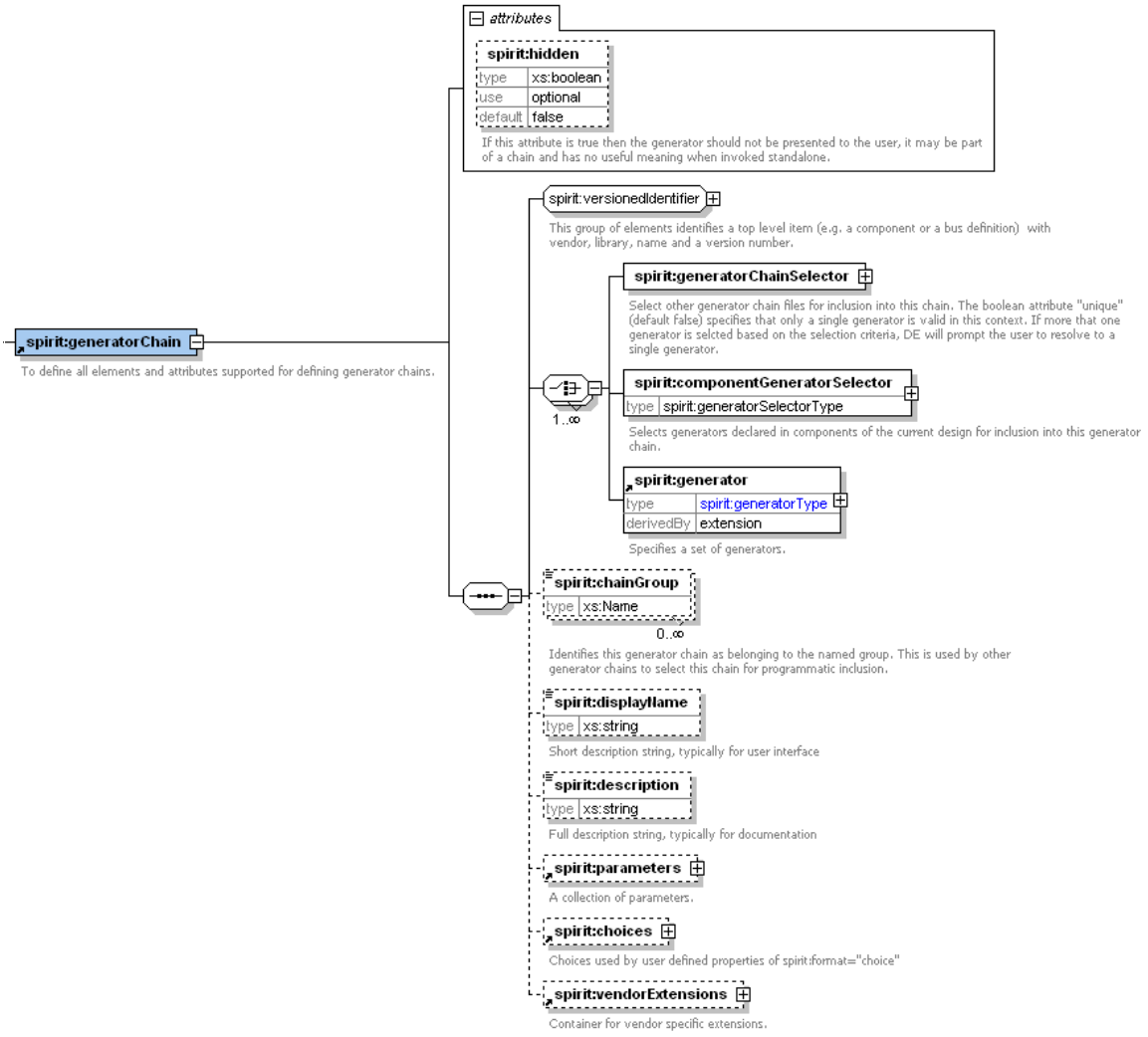
```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:generatorChain xmlns:spirit=http://www.spiritconsortium.org/
    XMLSchema/SPIRIT/1.4 xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4
    http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4/index.xsd">
   <spirit:vendor>spiritconsortium.org</spirit:vendor>
   <spirit:library>buildChain</spirit:library>
   <spirit:name>commonInit</spirit:name>
   <spirit:version>r1.0</spirit:version>
   <spirit:generator>
      <spirit:name>initNetlistGenerator</spirit:name>
      <spirit:phase>100</spirit:phase>
      <spirit:accessType>
         <spirit:readOnly>true</spirit:readOnly>
         <spirit:hierarchical>true</spirit:hierarchical>
         <spirit:instanceRequired>true</spirit:instanceRequired>
      </spirit:accessType>
<spirit::generatorExe>/user/spirit/generators/setupNetlist
</spirit::generatorExe>
   </spirit:generator>
   <spirit:componentGeneratorSelector>
      <spirit:groupSelector>
         <spirit:name>SPIRITGEN_SIM_HS_INIT</spirit:name>
      </spirit:groupSelector>
   </spirit:componentGeneratorSelector>
   <spirit:busGeneratorSelector>
      <spirit:groupSelector>
         <spirit:name>SPIRITGEN_SIM_HS_INIT</spirit:name>
      </spirit:groupSelector>
   </spirit:busGeneratorSelector>
   <spirit:chainGroup>SPIRITGEN_SIM_HS_INIT</spirit:chainGroup>
</spirit:generatorChain>
```

10.4 Generator schema

10.4.1 generatorChain

10.4.1.1 Schema

The following schema defines the information contained in the **generatorChain** top object.



10.4.1.2 Description

In IP-XACT, a design flow can be represented as a generator chain that links an ordered sequence of named tasks. Each named task can be represented as a single generator or as a generator chain. This way, design flow hierarchies can be constructed and executed from within a given DE. The DE itself is responsible for understanding the semantics of the specified chain described in the **generatorChain** XML.

General comment -- It would be nice if there was an indication of each entry as an element or an attribute.

The **generatorChain** element contains the following elements and attributes.

a) Mandatory elements

1) **versionedIdentifier** is a unique VLNV identifier.

2) At least one selector used to invoke a generator. A generator can be one of the following.

i) **generatorChainSelector** is a reference to another **generatorChain** (see 10.4.2).

ii) **componentGeneratorSelector** is a reference to a (list of) component generators (see 10.4.3).

iii) **generator** defines the generator (see 10.4.4).

b) Optional elements and attributes

1) **hidden** indicates (when set to *True*) this **generatorChain** object shall not be presented to the user (the default is *False*). For example, this may be part of a chain and have no useful meaning when invoked standalone.

2) **chainGroup** defines the list of ordered generator group names. This can be viewed as the list of events to which this **generatorChain** is sensitive.

3) In addition, a **generatorChain** can be further configured by specifying **parameters** and a set of **choices**. Lastly, its description can be enhanced by adding a **displayName** and **description** or extended using **vendorExtensions**. These generic elements can be found in other top IP-XACT objects, such as the component, and will therefore not be described here.
**Add xref OR copy that material here??
Let's copy this material as much as possible (and minimize this type of cross-reference

10.4.1.3 Example

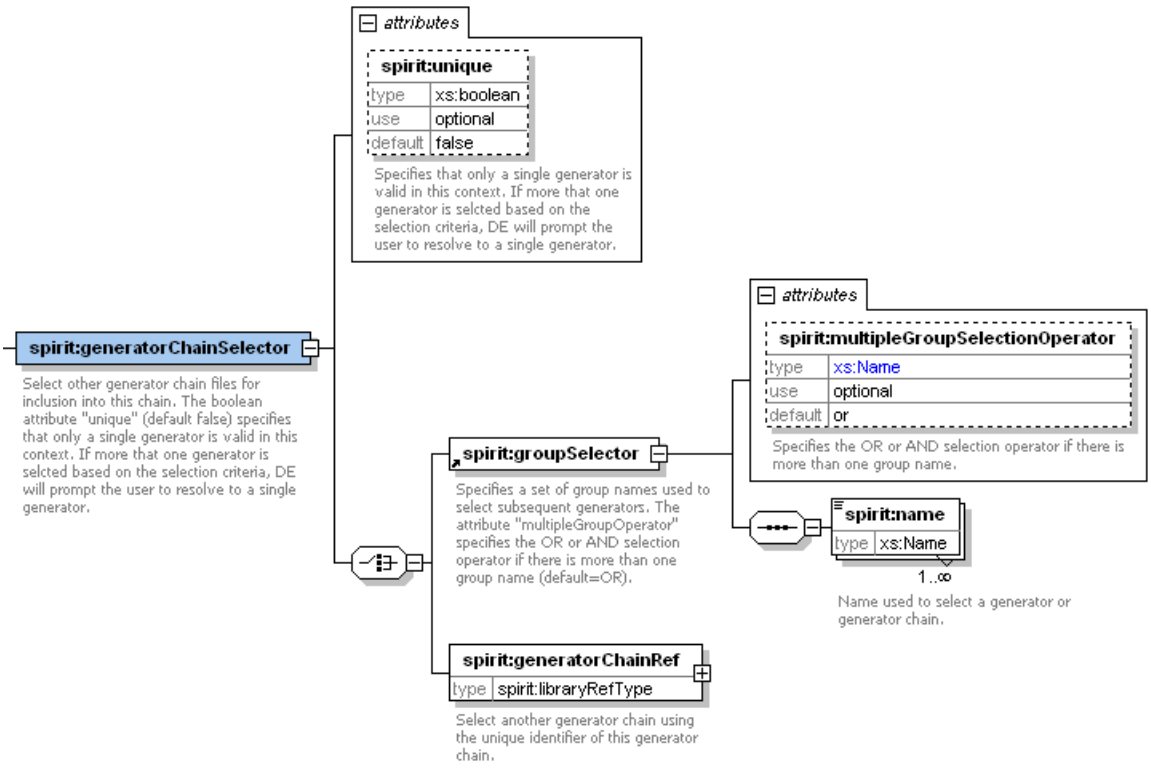
The following example defines a generator chain called `GEN_COSIM_CHAIN`, which is intended to specify a sequence of four simulation tasks (**INIT**, **CONFIG**, **BUILD**, and **COMPILE**) for both **HW** and **SW** compilation.

```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:generatorChain
xmlns:xs=http://www.w3.org/2001/XMLSchema
xmlns:spirit=http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4
    xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4
    http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4/index.xsd">
     <spirit:vendor>spiritconsortium.org</spirit:vendor>
     <spirit:library>buildChain</spirit:library>
     <spirit:name>CompleteBuild</spirit:name>
     <spirit:version>1.0</spirit:version>
     <spirit:generatorChainSelector>
         <spirit:groupSelector>
             <spirit:name>GEN_COSIM_INIT</spirit:name>
         </spirit:groupSelector>
     </spirit:generatorChainSelector>
     <spirit:generatorChainSelector>
         <spirit:groupSelector>
             <spirit:name>GEN_COSIM_CONFIG</spirit:name>
         </spirit:groupSelector>
     </spirit:generatorChainSelector>
     <spirit:generatorChainSelector>
         <spirit:groupSelector>
             <spirit:name>GEN_COSIM_BUILD</spirit:name>
         </spirit:groupSelector>
     </spirit:generatorChainSelector>
     <spirit:generatorChainSelector>
         <spirit:groupSelector>
             <spirit:name>GEN_COSIM_COMPILE</spirit:name>
         </spirit:groupSelector>
     </spirit:generatorChainSelector>
     <spirit:chainGroup>GEN_COSIM_CHAIN</spirit:chainGroup>
</spirit:generatorChain>
```

10.4.2 generatorChain selector

10.4.2.1 Schema

The following schema defines the information contained in the **generatorChainSelector** element, which may appear within a **generatorChain**.



10.4.2.2 Description

The **generatorChainSelector** element defines which generator(s) to invoke. This element contains the following mandatory elements and attributes.

a) **unique** specifies (when set to *True*) only a single generator can be selected (the defaults is *False*). If more that one generator is selected based on the selection criteria, the DE shall prompt the user to resolve to a single generator.

b) The selected generator(s) can be a **generatorChain** (referenced by its VLNV through the **generatorChainRef** element) or a list of group names (referenced by the **groupSelector**/**name** element) which identifies a list of generators (whose name match the given **groupSelector name**s).

The matching generators are the **generatorChain** generators whose **chainGroup** element values match one (or all if the **multipleGroupSelector** is set to **AND**) of the given **groupSelector name**s.

c) The **groupSelector** can be a single name or a list of names. When a list of names is specified, the **multipleGroupSelectorOperator** attribute can specify if the selection applies when one of the generator names matches (Boolean **OR**) or all the generator names match (Boolean **AND**).

10.4.2.3 Example

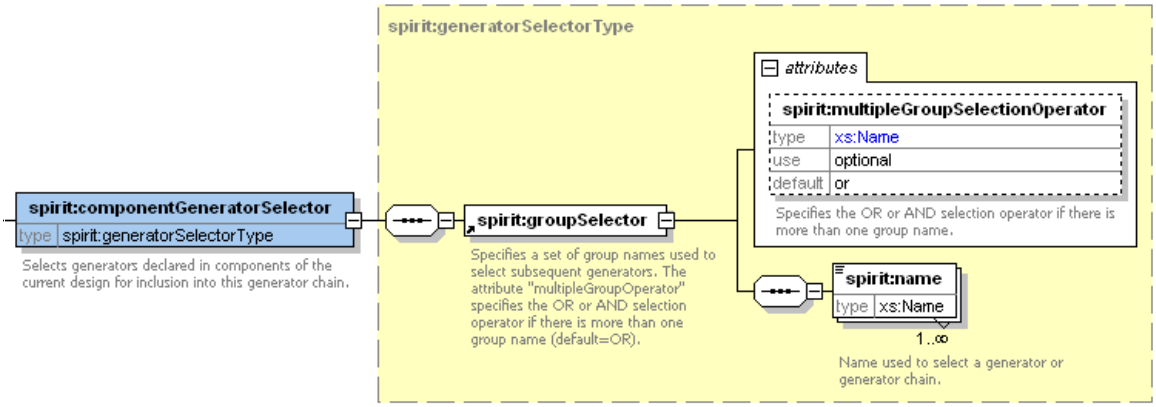
Assume three **generatorChain**s `X`, `Y`, and `Z` have been created with the **chainGroup** names `{A, B}`, `{A, C}`, and `{B, C}`, respectively. This example shows how a new **generatorChain** object can select `Y`.

```
<spirit:generatorChainSelector>
                    <spirit:groupSelector
   spirit:multipleGroupSelectionOperation="and">
                        <spirit:name>A</spirit:name>
                        <spirit:name>C</spirit:name>
                    </spirit:groupSelector>
</spirit:ports>
```

10.4.3 generatorChain component selector

10.4.3.1 Schema

The following schema defines the information contained in the **componentGeneratorSelector** element, which may appear within a **generatorChain**.



10.4.3.2 Description

Similar to the **generatorChainSelector**, **componentGeneratorSelector** selects a component generator or a list of component generators from a group selector. The following also apply.

a) The **groupSelector** can be a single name or a list of names. When a list of names is specified, the **multipleGroupSelectorOperator** attribute can specify if the selection applies when one of the generator names matches (Boolean **OR**) or all the generator names match (Boolean **AND**).

b) The matching generators are the component generators whose **groupName** element values match one (or all if the **multipleGroupSelector** is set to **AND**) of the **generatorChain**/**groupSelector** names.

10.4.3.3 Example

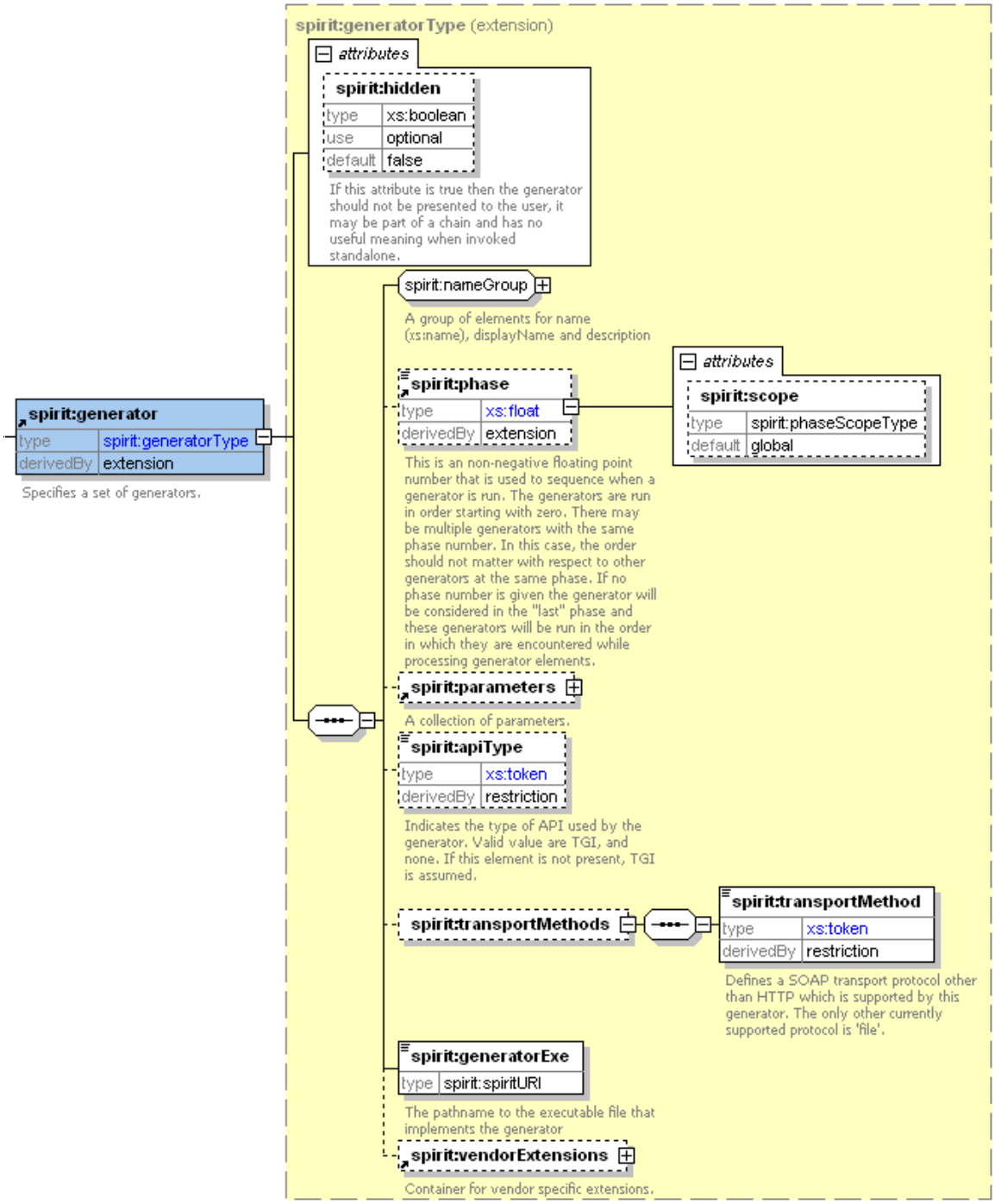
The following example shows a **generatorChain** selecting all the component generators whose **groupName** matches the name `docGen`.

```
<spirit:componentGeneratorSelector>
                    <spirit:groupSelector>
                        <spirit:name>docGen</spirit:name>
                    </spirit:groupSelector>
</spirit:componentGeneratorSelector>
```

10.4.4 generatorChain generator

10.4.4.1 Schema

The following schema defines the information contained in the **generator** element, which may appear within a **generatorChain** or **component**.



10.4.4.2 Description

The **generator** element describes a specific generator executable. This element contains the following elements and attributes.

a) Mandatory elements

1) **name** (included in ***nameGroup***) identifies the generator. ***nameGroup*** can also have two additional (optional) subelements: **displayName**, which allows a short descriptive text to be associated with the generator, and **description** which allows a textual description of the generator.

2) **generatorExe** defines an executable path, which shall include the command to launch this generator.

b) Optional information

1) **hidden** indicates (when set to *True*) this **generator** object shall not be presented to the user (the default is *False*). For example, this may be part of a chain and have no useful meaning when invoked standalone.

2) **phase** defines the sequence in which generators should be fired. In addition, the **scope** attribute can be used to attach a generator phase: **local** or **global** (default).

3) **parameters** defines the generator parameters described as a list of name and value pairs, which can be extended with specific **vendorAttributes** or **vendorExtensions**.

4) **apiType** is the API used by the generator: **TGI** (the default) or **None** (to designate there is no communication between the DE and the generator).

5) **transportMethods** defines the list of transport protocols (other than `http`) supported by this generator. The only supported protocol is **file**.

6) **vendorExtensions** adds any extra vendor-specific data related to the generator.

10.4.4.3 Example

The following example shows a netlist generator.

```
<spirit:generator>
                    <spirit:name>generateNetlist<spirit:name>
                    <spirit:phase>100.0</spirit:phase>
                    <spirit:parameters>
                        <spirit:parameter>
                            <spirit:name>language<spirit:name>
                            <spirit:value
                                spirit:id=netlistGenLangId
                                spirit:resolve=user
                                spirit:choiceRef= netlistGenLangChoicesId>vhdl
                            </spirit:value>
                        </spirit:parameter>
                    </spirit:parameters>
                    <spirit:apiType>TGI</spirit:apiType>
                    <spirit:generatorExe>tclsh ../generic_netlister.tcl</
    spirit:generatorExe>
</spirit:generator>
```

11. Design configuration descriptions

11.1 Design configuration

IP-XACT includes a schema for documents that store design configuration information—all the configurable information that is not recorded in the design file. The design configuration information is useful when transporting designs between design environments; it contains information that would otherwise have to be re-entered by the designer; while the *design itself* contains all information regarding configuration of the design, e.g., instance base addresses.

The *design configuration file* contains the following configuration information.

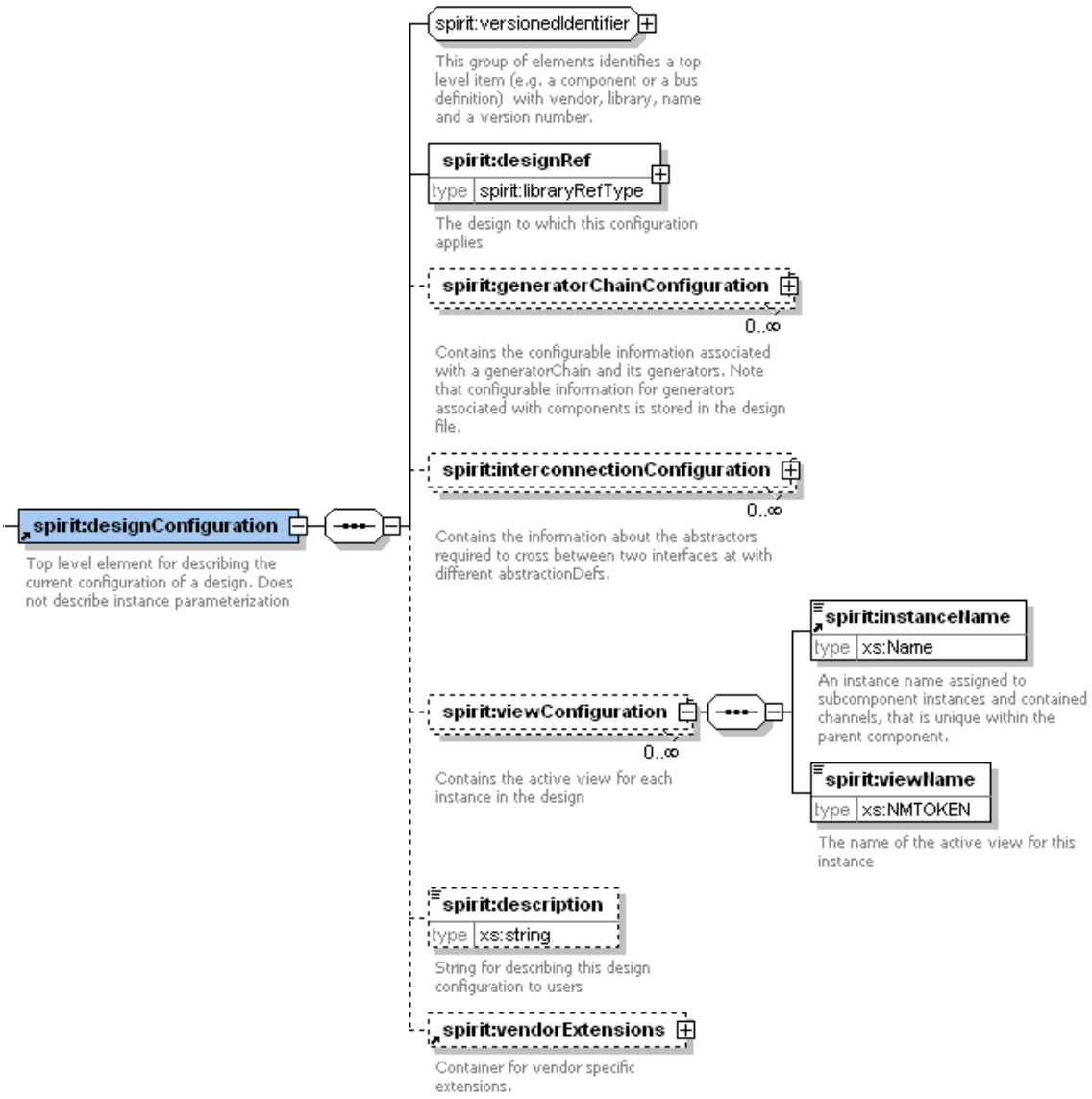
- configurable information defined in generators within generator chains; this information is not referenced via the design file;
- the active, or current, view selected for instances in the design;
- the configuration information for interconnections between the same bus types with differing abstraction types (i.e., abstractor reference, parameter configuration, and view selection). See also: the abstractor section 4.9.2.

Finally, a design configuration applies to a single design, but a design may have multiple design configuration files.

11.2 designConfiguration

11.2.1 Schema

The following schema defines the information contained in the **designConfiguration** root element.



11.2.2 Description

The **designConfiguration** element details the configuration for a design. It contains the following mandatory and options elements.

a) Mandatory elements

1) **versionedIdentifier** is a group containing the **vendor**, **library**, **name**, and **version** elements.

2) **designRef** specifies the design VLNV to which the configuration applies. It has the **vendor**, **library**, **name**, and **version** attributes.

b) Optional elements

1) **generatorChainConfiguration** contains the configurable information associated with a generator defined within a **generatorChain**. See 11.3.

2) **interconnectionConfiguration** contains information about the abstractors required for the connection of two interfaces with different **abstractionDefinition** types. See 11.4.

3) **viewConfiguration** lists the active view for each instance of the design. It has the following subelements.

i) **instanceName** specifies the component instance name for which the view is being selected. This instance name shall be unique with other instance names inside the referenced design file.

ii) **viewName** defines the current valid view for the selected component instance.

4) **description** allows a textual description of the design configuration.

5) **vendorExtensions** adds any extra vendor-specific data related to the design configuration.

See also: SCR 1.5.

11.2.3 Example

The following example shows a **designConfiguration** containing a generator chain configuration: one abstractor configuration in an **interconnectionConfiguration** and one instance view configuration.

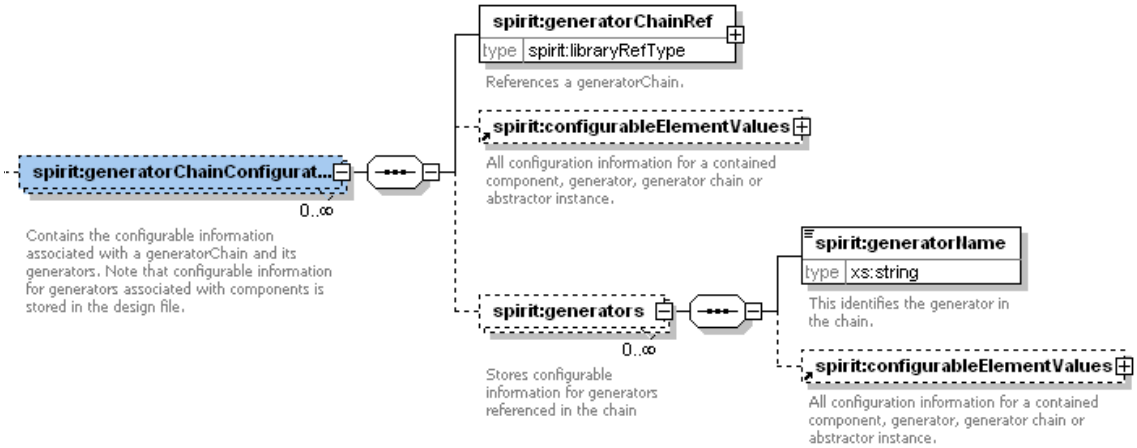
```
<spirit:designConfiguration xmlns:spirit="http://www.spiritconsortium.org/
   XMLSchema/SPIRIT/1.4" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
   instance" xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/
   SPIRIT/1.4/index.xsd">
  <spirit:vendor>spiritconsortium.org</spirit:vendor>
  <spirit:library>Library</spirit:library>
  <spirit:name>Configs</spirit:name>
  <spirit:version>1.0</spirit:version>
  <spirit:designRef spirit:vendor="spiritconsortium.org"
   spirit:library="DesignLibrary" spirit:name="Design1"
   spirit:version="1.0"/>
  <spirit:generatorChainConfiguration>
      <spirit:generatorChainRef spirit:vendor="spiritconsortium.org"
   spirit:library="generatorLibrary" spirit:name="generator1"
   spirit:version="1.0"/>
      <spirit:generators>
          <spirit:generatorName>gen1</spirit:generatorName>
                      <spirit:configurableElementValues>
                         <spirit:configurableElementValue
   spirit:referenceId="tmpDir">           my_temp_dir</
   spirit:configurableElementValue>
                      </spirit:configurableElementValues>
                   </spirit:generators>
  </spirit:generatorChainConfiguration>
  <spirit:interconnectionConfiguration>
    <spirit:interconnectionRef>
      connection1
    </spirit:interconnectionRef>
    <spirit:abstractors>
      <spirit:abstractor>
        <spirit:instanceName>a1</spirit:instanceName>
          <spirit:abstractorRef
             spirit:vendor="spiritconsortium.org"
             spirit:library="AbstractorLibrary"
             spirit:name="AHBPvToRtl"
             spirit:version="1.0"/>
        <spirit:viewName>verilog</spirit:viewName>
      </spirit:abstractor>
</spirit:abstractors>
  </spirit:interconnectionConfiguration>
  <spirit:viewConfiguration>
    <spirit:instanceName>instance_1</spirit:instanceName>
    <spirit:viewName>verilog</spirit:viewName>
```

```
    </spirit:viewConfiguration>
  </spirit:designConfiguration>
```

11.3 generatorChainConfiguration

11.3.1 Schema

The following schema defines information contained in **generatorChainConfiguration**, which may appear as an element inside the **designConfiguration** root element.



11.3.2 Description

The **generatorChainConfiguration** element contains the configurable information associated with a **generatorChain** and its generators. Configurable information for any generators associated with components is stored in the design file (in the **configuration** of an instance associated with a **componentGenerator**). The **generatorChainConfiguration** element contains the following mandatory and options elements.

a) Mandatory elements

generatorChainRef points to the VLNV of a **generatorChain** through the **vendor**, **library**, **name**, and **version** attributes.

b) Optional elements

generators specify any configurable information for the generators referenced in a chain. It has the following subelements.

i) **generatorName** (mandatory) identifies the generator in the referenced chain.

ii) **configurableElementValues** (optional) specifies any **configurableElementValue** elements, which contain values for the generator configurable elements, referenced via the mandatory **referenceId** attribute.

See also: SCR 1.7.

11.3.3 Example

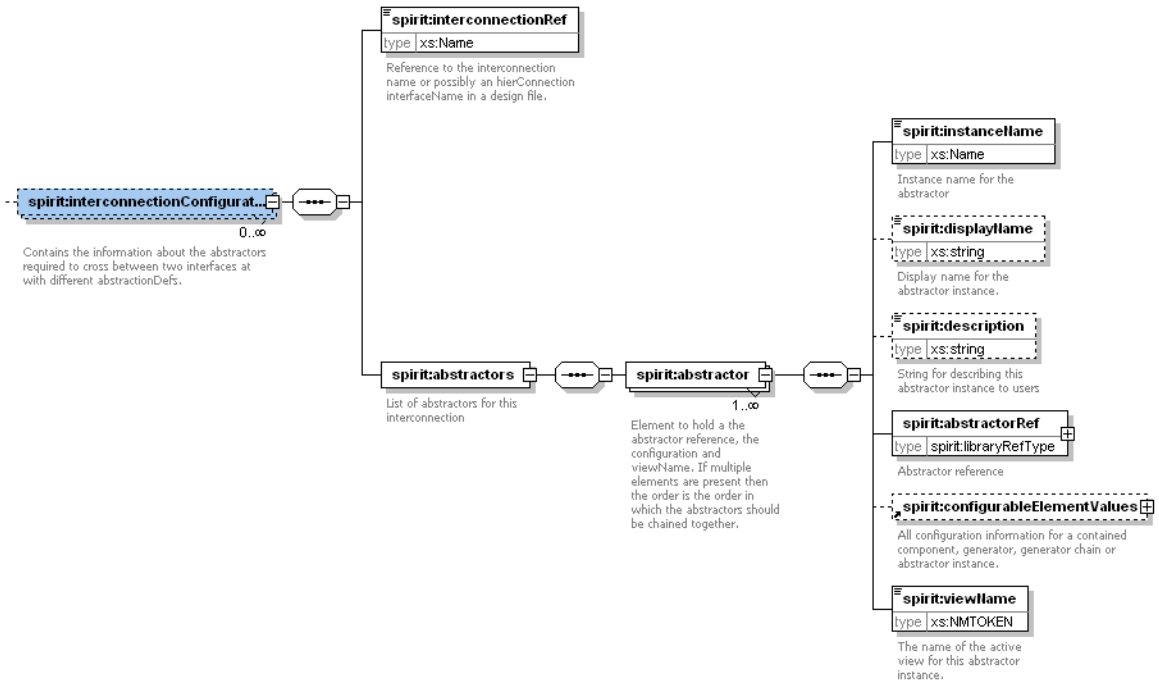
The following example shows the configurable information for a **generatorChain**. Here two generators inside the referenced **generatorChain** are configured.

```
<spirit:generatorChainConfiguration>
      <spirit:generatorChainRef spirit:vendor="spiritconsortium.org"
   spirit:library="generatorLibrary" spirit:name="generator1"
   spirit:version="1.0"/>
      <spirit:generators>
          <spirit:generatorName>gen1</spirit:generatorName>
                     <spirit:configurableElementValues>
                        <spirit:configurableElementValue
   spirit:referenceId="tmpDir">           my_temp_dir</
   spirit:configurableElementValue>
                  </spirit:generators>
                  <spirit:generators>
          <spirit:generatorName>gen2</spirit:generatorName>
                     <spirit:configurableElementValues>
                        <spirit:configurableElementValue
   spirit:referenceId="verbose_level">           1</
   spirit:configurableElementValue>
                      <spirit:configurableElementValue
   spirit:referenceId="dump_log">           true</
   spirit:configurableElementValue>
                     </spirit:configurableElementValues>
                  </spirit:generators>
  </spirit:generatorChainConfiguration>
```

11.4 interconnectionConfiguration

11.4.1 Schema

The following schema defines information contained in **interconnectionConfiguration** element, which may appear as an element inside the **designConfiguration** root element.



11.4.2 Description

The **interconnectionConfiguration** element contains information about the abstractors used to connect two interfaces having the same **busDefinition** types and different **abstractionDefinition** types. The **interconnectonConfiguration** element contains the following mandatory elements and attributes.

a) **interconnectionRef** contains a reference to a design interconnection name or a **hierConnection interfaceRef** name.

b) **abstractors** contains the **abstractor** elements, this list of elements specify the order in which the abstractors shall be chained together to bridge from one abstraction to another. An **abstractor** has the following subelements.

1) **instanceName** (mandatory) defines the name of the abstractor instance.

2) **abstractorRef** (mandatory) points to the VLNV of the **abstractor** through the **vendor**, **library**, **name**, and **version** attributes.

3) **viewName** (mandatory) defines the name of the active view for this abstractor instance.

4) **displayName** (optional) defines the display name for the abstractor instance.

5) **description** (optional) provides a textual description of the abstractor instance.

6) **configurableElementValues** (optional) has **configurableElementValue** elements, which describe the values of configurable elements of the referenced **generatorChain**. The mandatory **referenceId** attribute in a **configurableElementValue** specifies the **id** of the configurable element to reconfigure.

General comment -- section 11.4.2 includes all sub-elements in a single list and then indicates within each list entry whether or not the entry is optional. This is inconsistent with the way chapter 10 was done where the mandatory and optional elements are separated into different lists. I like the chapter 11 approach better because it allows for closer alignment with the schema pictures, but the important thing is to be consistent.

See also: SCR 3.13, SCR 3.14, SCR 3.15, SCR 3.16, SCR 3.17, SCR 3.18, SCR 3.19, SCR 3.20, SCR 3.21, and SCR 3.22.

11.4.3 Example

The following example shows the configuration of the `connection1` interconnection, with the definition of a chain of two abstractors to insert to bridge the two abstractions. The abstractor instances are `abstraction1` and `abstraction2`. The active views of these abstractor instances are `verilog` and `verilog_view`. The abstractor VLNVs are defined in the **abstractorRef** elements.

```
<spirit:interconnectionConfiguration>
    <spirit:interconnectionRef>
      connection1
    </spirit:interconnectionRef>
    <spirit:abstractors>
      <spirit:abstractor>
        <spirit:instanceName>abstractor1</spirit:instanceName>
        <spirit:abstractorRef
             spirit:vendor="spiritconsortium.org"
             spirit:library="AbstractorLibrary"
             spirit:name="AHBPvToAHBPvt"
             spirit:version="1.0" />
        <spirit:viewName>verilog</spirit:viewName>
      </spirit:abstractor>
                  <spirit:abstractor>
        <spirit:instanceName>abstractor2</spirit:instanceName>
        <spirit:abstractorRef
             spirit:vendor="spiritconsortium.org"
```

```
            spirit:library="AbstractorLibrary"
            spirit:name="AHBPvtToRtl"
            spirit:version="1.0" />
          <spirit:viewName>verilog_view</spirit:viewName>
        </spirit:abstractor>
  </spirit:abstractors>
    </spirit:interconnectionConfiguration>
```

12. Addressing and addressing formulas

Once the WG approves the technical content of Anthony's IPXACTaddressing.doc write-up, I'll incorporate it into this clause. In the meantime, this merely serves as a placeholder.

Annex A

(informative)

Bibliography

Update this list as relevant

[B1] Bradner, S., IETF RFC 2119 *"Key words for use in RFCs to Indicate Requirement Levels."* Best Current Practice: 14 (See http://www.ietf.org/rfc/rfc2119.txt.)

[B2] IEEE 100, *The Authoritative Dictionary of IEEE Standards Terms,* Seventh Edition. New York: Institute of Electrical and Electronics Engineers, Inc.

[B3] *IP-XACT Leon Register Transfer Examples*, v1.4, see http://www.spiritconsortium.org/doc_downloads/???. **add the actual reference**

[B4] *IP-XACT Leon Transaction Level Examples*, v1.4, see http://www.spiritconsortium.org/doc_downloads/???. **add the actual reference**

[B5] *IP-XACT Schema on-line documentation*, v1.4, see http://www.spiritconsortium.org/doc_downloads/???. **add the actual reference**

[B6] *IP-XACT Tight Generator Interface Overview*, v1.4, see http://www.spiritconsortium.org/doc_downloads/???. **add the actual reference**

[B7] The Transaction Level Model of SystemC. This model is in the process of standardization by the Open SystemC Initiative (OSCI) (http://www.systemc.org)

Annex B

(normative)

Semantic consistency rules

Generally, any "i.e." additions should be in the Rules column, not the Notes column; let's confirm all these before rearranging their placement in each table

For an IP-XACT document or a set of IP-XACT documents, to be valid they shall, in addition to conforming to the IP-XACT schema, obey certain semantic rules. While many of these are described informally in other sections of this document, this chapter defines them formally. Tools generating IP-XACT documents must ensure these rules are obeyed. Tools reading IP-XACT documents shall report any breaches of these rules to the user.

Most of the semantic rules listed here can be checked purely by manually examining a set of IP-XACT documents. A few, listed at the end of this annex, need some external knowledge, so they cannot be checked this way. In Table B1 — Table B14, *Single doc check* indicates a rule can be checked purely by manually examining a single IP-XACT document. Rules for which *Single doc check* is `No` require the examination of the relationships between *IP-XACT* documents.

Table B1—Cross-references and VLNVs

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 1.1	1	Every IP-XACT document visible to a tool shall have a unique VLNV.	No	Only applies only to those documents visible to a particular tool or DE at one time. In particular, users are likely to store multiple versions of the same documents, with the same VLNVs, in source control systems.
SCR 1.2	2	Any VLNV in an IP-XACT document used to reference another IP-XACT document shall precisely match the identifying VLNV of an existing IP-XACT document.	No	In the schema, such references always use the attribute group **versionedIdentifier**.
SCR 1.3	3	The VLNV in an **extends** element in a bus definition shall be a reference to a bus definition.	No	
SCR 1.4	4	The VLNV in a **busType** element in a bus interface or abstraction definition shall be a reference to a bus definition.	No	
SCR 1.5	5	The VLNV in a **designRef** element in a design configuration shall be a reference to a **design**.	No	

Table B1—Cross-references and VLNVs (Continued)

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 1.7	7	The VLNV in a **generatorChainRef** element in a design configuration shall be a reference to a generator chain.	No	
SCR 1.9	9	The VLNV in a **generatorChainRef** sub-element of the element **generatorChainSelector** in a generator chain shall be a reference to a generator chain.	No	
SCR 1.11	11	The VLNV in a **componentRef** element in a **design** shall be a reference to a **component**.	No	
SCR 1.12		The XML document element of a an IP-XACT document shall be an **abstractor**, **abstractionDefinition**, **busDefinition**, **component**, **design**, **designConfiguration** or **generatorChain** element.	No	
SCR 1.13		The VLNV in an **abstractionType** element in a component or abstractor shall reference an **abstractionDefiniton**.		
SCR 1.14		If a bus definition contains an **abstractionType** sub-element, the abstraction definition's **busType** element and the bus interface's **busType** element shall reference the same bus definition.	No	I.e., the abstraction referenced shall be an abstraction of the referenced bus.
SCR 1.15		The VLNV in an **abstractorRef** in a **designConfiguration** shall reference an **abstractor**.	No	
SCR 1.16		The VLNV in an **extends** element in an abstraction definition shall be a reference to an abstraction definition.	No	

Table B2—Interconnections

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 2.1	12.	In the attributes of an **activeInterface** or **monitorInterface** element, the value of the **busRef** attribute shall be the name of a **busInterface** in the component description referenced by the VLNV of the component instance named in **componentRef** attribute.	No	
SCR 2.2	13.	In the sub-elements of an **interconnection** element, the bus interfaces referenced by the two **activeInterface** sub-elements shall be compatible, i.e., the VLNVs of the **busType** elements within the two **busInterface** elements shall reference compatible **busDefinitions**.	No	

Table B2—Interconnections (Continued)

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 2.3	14.	A particular component/bus interface combination shall appear in only one **interconnection** element in a design.	Yes	
SCR 2.4	15.	An **interconnection** element shall only connect a master interface to a slave interface or a mirrored-master interface.	No	
SCR 2.5	16.	An **interconnection** element shall only connect a mirrored-master interface to a master interface.	No	
SCR 2.6	17.	An **interconnection** element shall only connect a slave interface to a master interface or a mirrored-slave interface.	No	
SCR 2.7	18.	An **interconnection** element shall only connect a mirrored-slave interface to a slave interface.	No	
SCR 2.8	19.	An **interconnection** element shall only connect a direct system interface to a mirrored-system interface.	No	
SCR 2.9	20.	An **interconnection** element shall only connect a mirrored-system interface to a direct system interface.	No	
SCR 2.10	21.	In a direct master to slave connection, the value of **bitsInLAU** in the master's address space shall match the value of **bitsInLAU** in the slave's memory map.	No	
SCR 2.11	22.	In a direct master to slave connection, the range of the master's address space shall be greater or equal to the range of the slave's memory map.	No	When the slave's memory map is defined in terms of memory banks or subspace maps, calculating its range may be complex.
SCR 2.12	23.	In a direct master to slave connection, the **busDefinitions** referenced by the **busInterfaces** shall have a **directConnection** element with the value *True*.	No	
SCR 2.13	24.	In a connection between a system interface and a mirrored-system interface, the values of the **group** elements of the two bus interfaces shall be identical.	No	
SCR 2.14		If the same logical port is in the port map of both ends of a direct master to slave connection, the **vector** elements of that logical port shall be identical in the two port maps.	No	Logical ports can only be identified with one another if the two bus interfaces reference the same abstraction definition.

Delete these definitions and/or move them into the appropriate Schema section

B.0.1 Compatibility of busDefinitions

a) A **busDefinition** `A` is an extension of **busDefinition** `B` if `A` contains an extension element that references `B` or an extension of `B`.

b) A **busDefinition** is compatible with itself.

c) If `A` is an extension of `B`, then `A` and `B` are compatible.

d) No other pairs of **busDefinition**s are compatible.

e) A set of **busDefinition**s `{A, B, C, ...}` is compatible if every possible pair of **busDefinition**s from the set (`{ A, B }`, `{ A, C }`, `{ B, C }` ...) is compatible.

B.0.2 Interface mode of a bus interface

Specifies whether the bus interface is a **master**, **slave**, **system**, **mirroredMaster**, **mirroredSlave**, **mirroredSystem**, or **monitor** interface.

Table B3—Channels, bridges, and abstractors

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 3.1	25.	Within a **channel** element, all the **busInterfaceRef** elements shall refer to compatible abstraction definitions, i.e., the VLNVs of the **abstractionType** elements within the **busInterface** elements shall reference compatible **abstractionDefinition**s.	No	Compatibility of the abstraction definitions implies compatibility of their associated bus definitions.
SCR 3.2	26.	All bus interfaces referenced by a channel shall be mirrored interfaces.	Yes	
SCR 3.3	27.	A channel can be connected to no more mirrored-master **busInterface**s than the least value of **maxMasters** in the **busDefinition**s referenced by the connected **busInterface**s (whether these interfaces are mirrored-master or mirrored-slave interfaces).	No	A channel may connect ports with different bus definitions, and hence different values of **maxMasters**, as long as the bus definitions are compatible.
SCR 3.4	28.	A channel can be connected to no more mirrored-slave bus interfaces than the least value of **maxSlaves** in the bus definitions referenced by the connected bus interfaces (whether these interfaces are mirrored-master or mirrored-slave interfaces).	No	A channel may connect ports with different bus definitions, and hence different values of **maxSlaves**, as long as the bus definitions are compatible.
SCR 3.5	29.	Each bus interface on a component shall connect to only one channel of that channel component.	Yes	

Table B3—Channels, bridges, and abstractors (Continued)

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 3.6	30.	The interface referenced by **masterRef** sub-element of a **bridge** element shall be a master.	Yes	
SCR 3.13		The value of the **interconnectionRef** sub-element of an **interconnectionConfiguration** element shall precisely match the name of an interconnection described in the design referenced by the containing design configuration.	No	
SCR 3.14		An **interconnectionConfiguration** element of a design configuration document that references a master to mirrored-master interconnection in the corresponding design shall only reference abstractors with an **abstractorMode** of **master**.	No	
SCR 3.15		An **interconnectionConfiguration** element of a design configuration document that references a slave to mirrored-slave interconnection in the corresponding design shall only reference abstractors with an **abstractorMode** of **slave**.	No	
SCR 3.16		An **interconnectionConfiguration** element of a design configuration document that references a system to mirrored-system interconnection in the corresponding design shall only reference abstractors with an **abstractorMode** of **system**.	No	
SCR 3.17		An **interconnectionConfiguration** element of a design configuration document that references a master to slave interconnection in the corresponding design shall only reference abstractors with an **abstractorMode** of **direct**.	No	
SCR 3.18		An **interconnectionConfiguration** element shall not reference an interconnection in which the abstraction types referenced by the two endpoints are identical.	No	
SCR 3.19		In the list of abstractors referenced by an **interconnectionConfiguration** element, the first **abstractionType** element of the first referenced abstractor shall be compatible with the **abstractionType** element of the master, system, or mirrored-slave endpoint of the interconnection.	No	Rules 3.19 — 3.22 mean the abstractors associated with an interconnection need to form a non-looping chain between the two ends.
SCR 3.20		In the list of abstractors referenced by an **interconnectionConfiguration** element, the second **abstractionType** element of the last referenced abstractor shall be compatible with the **abstractionType** element of the mirrored-master, mirrored-system, or slave endpoint of the interconnection.	No	

Table B3—Channels, bridges, and abstractors (Continued)

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 3.21		In the list of abstractors referenced by an **interconnectionConfiguration** element, the first **abstractionType** element of every referenced abstractor, except the first, shall be compatible with the second **abstractionType** element of the previous abstractor in the **interconnectionConfiguration** list.	No	
SCR 3.22		In the list of abstractors referenced by an **interconnectionConfiguration** element, no two **abstractionType** elements in the referenced abstractors shall have the same value.	No	
SCR 3.23		The VLNVs in the **busType** elements of both abstraction definitions referenced by an abstractor shall exactly match the VLNV in the **busType** element of the abstractor.	No	
SCR 3.24		If abstraction definition `AA` is an abstraction of bus definition `A` and abstraction definition `AB` is an abstraction of bus definition `B`, then abstraction definition `AA` shall only contain an **extension** element referencing abstraction definition `AB` if bus definition `A` contains an **extension** element referencing bus definition `B`.	No	If abstraction definition `AA` extends abstraction definition `AB`, `AA` and `AB` need to be abstractions of different buses.

Delete these definitions and/or move them into the appropriate Schema section

B.0.3 Compatibility of abstractionDefinitions

a) An **abstractionDefinition** `A` is an extension of **abstractionDefinition** `B` if `A` contains an extension element that references `B` or an extension of `B`.

b) An **abstractionDefinition** is compatible with itself.

c) If `A` is an extension of `B`, then `A` and `B` are compatible.

d) No other pairs of **abstractionDefinition**s are compatible.

Table B4—Monitor interfaces and interconnections

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 4.1	31.	An **interconnection** element cannot reference a monitor interface.	No	
SCR 4.2	32.	The **activeInterface** sub-element of a **monitorInterconnection** element shall reference a **master**, **slave**, **system**, **mirroredMaster**, **mirroredSlave**, or **mirroredSystem** interface.	No	
SCR 4.3	33.	The **monitorInterface** sub-elements of a **monitorInterconnection** element shall reference a **monitor** bus interface.	No	
SCR 4.4	34.	In a **monitorInterconnection** element, the value of the **interfaceModeMode** of the monitor interfaces shall match the **interfaceModeMode** of the active interface.	No	This means all the active interfaces shall have the same interface mode.
SCR 4.5	35.	A **monitor** interface shall only be connected to a **system** or **mirroredSystem** interface if it has a **group** sub-element and the value of this element matches the value of the **group** sub-element of the **system** or **mirroredSystem** interface.	No	
SCR 4.6	36.	A particular **component**/**busInterfaceName** combination shall only appear in one **monitorInterconnection** element.	No	This applies to both monitor and active interfaces; however, a single **monitorInterconnection** element can connect an active interface to many monitor interfaces. The same active interface can also appear in at most one **interconnection** element.

Table B5—Configurable elements

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 5.1	37.	A configurable element shall have a **dependency** attribute if and only if it has a **resolve** attribute with the value **dependent**.	Yes	
SCR 5.2	38.	The value of a **dependency** attribute shall be an XPATH expression. This XPATH expression shall only reference the containing document.	Yes	
SCR 5.3	39.	The XPATH expression in a **dependency** attribute shall not reference configurable elements having a **resolve** attribute value of **dependent** or **generated**.	Yes	
SCR 5.4	40.	Any parameters used within all dependent parameter's `XPATH id()` calls shall exist.	Yes	
SCR 5.5	41.	All references to elements in **dependency** XPATH expressions shall be by **id**. Dependency XPATH expressions shall not use document navigation to reference other elements.	Yes	This rule allows XPATH expressions to remain valid through schema or design changes. DEs reading IP-XACT documents should treat breaches of this rule as minor errors, and attempt to interpret any XPATH expressions in the document.
SCR 5.6	42.	An **id** attribute is required in any element with a **resolve** attribute value of **user** or **generated**.	Yes	
SCR 5.7	43.	**configurableElement** elements within **componentInstance** elements shall only reference configurable elements that exist in the **component** referenced by the enclosing **componentInstance** element; the value of the **referenceId** attribute of the **configurableElement** element shall match the value of the **id** attribute of some configurable element of the **component**.	No	The schema guarantees uniqueness of **id** values in a **component**.
SCR 5.8	44.	**configurableElement** elements shall only reference configurable elements with a **resolve** attribute value of **user** or **generated**.	No	
SCR 5.9	45.	If a **configurableElement** element references an element with a **formatType** attribute value of **float** or **long** and containing a **minimum** attribute, the value of the **configurableElementValue** element shall be greater or equal to the specified value of the **minimum** attribute.	No	

Table B5—Configurable elements (Continued)

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 5.10	46.	If a **configurableElement** element references an element with a **format** attribute value of **float** or **long** and containing a **maximum** attribute, the value of the **configurableElementValue** sub-element shall be less than or equal to the specified value of the **maximum** attribute.	No	
SCR 5.11	47.	If an element has a **format** attribute with a value of **choice**, it also needs a **choiceRef** attribute.	Yes	
SCR 5.12	48.	If a **configurableElement** element references an element with a **choiceRef** attribute, the value for **configurableElementValue** sub-element shall be one of the values listed in the **choice** element referenced by the **choiceRef** attribute.	No	
SCR 5.13	.	**configurableElement** elements within **generatorChain** elements in design configuration documents shall only reference configurable elements that exist in the generator chain referenced by the enclosing **generatorChain** element; the value of the **referenceId** attribute of the **configurableElement** element shall match the value of the **id** attribute of some configurable element of the generator chain.	No	The schema guarantees uniqueness of **id** values in a generator chain.
SCR 5.14	.	**configurableElement** elements within **generator** elements in design configuration documents elements shall only reference configurable elements that exist in the generator referenced by the enclosing **generator** element (within the generator chain referenced by the enclosing **generatorChain** element); the value of the **referenceId** attribute of the **configurableElement** element shall match the value of the **id** attribute of some configurable element of the generator.	No	The schema guarantees uniqueness of **id** values in a generator chain.
SCR 5.15	.	**configurableElement** elements within **abstractor** elements in design configuration documents elements shall only reference configurable elements that exist in the abstractor referenced by the enclosing **abstractor** element; the value of the **referenceId** attribute of the **configurableElement** element shall match the value of the **id** attribute of some configurable element of the abstractor.	No	The schema guarantees uniqueness of **id** values in an abstractor.

Delete these definitions and/or move them into the appropriate Schema section

B.0.4 Configurable element

This is an element that uses the **common.att** attribute group. The definition of such elements can define that its value is derived by calculation from other elements, or set by the user or a generator.

Note—This is different from a `configurableElement` element, which is an element that references and sets the value of a configurable element.

B.0.5 Element referenced by configurableElement element

Every **configurableElement** element references a component document and is contained within a **componentInstance** element. The element referenced by a **configurableElement** element is the configurable element in that component document with an **id** attribute matching the **referenceId** of the **configurableElement** element.

Table B6—Ports

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 6.1	49.	The value of any **busPortName** sub-element in a **busInterface** element shall match the value of a **logicalName** element of the abstraction definition referenced by the **busInterface** element.	No	
SCR 6.5.1		If the abstraction definition referenced by a bus interface specifies an initiative value for a logical port of **requires** for that interface mode of bus interface, the port map shall only map that logical port to a component port with an initiative value of **requires**, **both**, or **phantom**, or to a component port with an **allLogicalInitiativesAllowed** attribute with the value *True*. For system interfaces, the port initiative values shall be looked up from the **onSystem** element with the group name matching that of the bus interfaces. For mirrored interfaces, the bus port initiative values needs to be reversed before doing the comparison.	No	

Table B6—Ports (Continued)

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 6.5.2		If the abstraction bus definition referenced by a bus interface specifies an initiative value for a logical port of **provides** for that interface mode of bus interface, the port map shall only map that logical port to a component port with an initiative value of **provides**, **both**, or **phantom**, or to a component port with an **allLogicalInitiativesAllowed** attribute with the value *True*. For system interfaces, the port initiative values shall be looked up from the **onSystem** element with the group name matching that of the bus interfaces. For mirrored interfaces, the bus port initiative values shall be reversed before doing the comparison. Mirrored bus interfaces shall be looked up as if they were not mirrored.	No	
SCR 6.5.3		If the abstraction bus definition referenced by a bus interface specifies an initiative value for a logical port of **both** for that interface mode of bus interface, and the bus interface has a port map, the port map shall only map that logical port to a component port with an initiative value of **both** or **phantom**, or to a component port with an **allLogicalInitiativesAllowed** attribute with the value *True*. For system interfaces, the port initiative values shall be looked up from the **onSystem** element with the group name matching that of the bus interfaces. For mirrored interfaces, the bus port initiative values shall be reversed before doing the comparison. Mirrored bus interfaces shall be looked up as if they were not mirrored.	No	
SCR 6.6.1		If the abstraction definition referenced by a bus interface specifies a direction for a logical port of **in** for that interface mode of bus interface, the port map shall only map that logical port to a component port with a direction of **in**, **inout**, or **phantom**, or to a component port with an **allLogicalDirectionsAllowed** attribute with the value *True***.** For system interfaces, the port directions shall be looked up from the **onSystem** element with the group name matching that of the bus interfaces. For mirrored interfaces, the bus port directions shall be reversed before doing the comparison.	No	

Table B6—Ports (Continued)

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 6.6.2		If the abstraction definition referenced by a bus interface specifies a direction for a logical port of **out** for that interface mode of bus interface, the port map shall only map that logical port to a component port with a direction of **out**, **inout**, or **phantom**, or to a component port with an **allLogicalDirectionsAllowed** attribute with the value *True***.** For system interfaces, the port directions shall be looked up from the **onSystem** element with the group name matching that of the bus interfaces. For mirrored interfaces, the bus port directions shall be reversed before doing the comparison.	No	
SCR 6.6.3		If the abstraction definition referenced by a bus interface specifies a direction for a logical port of **inout** for that interface mode of bus interface, the port map shall only map that logical port to a component port with a direction of **inout** or **phantom**, or to a component port with an **allLogicalDirectionsAllowed** attribute with the value *True***.** For system interfaces, the port directions shall be looked up from the **onSystem** element with the group name matching that of the bus interfaces. For mirrored interfaces, the bus port directions shall be reversed before doing the comparison.	No	
SCR 6.7		If the abstraction definition referenced by a bus interface specifies, for a port, a presence value of **required** for that interface mode of bus interface, and the bus interface has a port map, the port shall be in that port map. For system interfaces, the port presence shall be looked up from the **onSystem** element with the group name matching that of the bus interfaces. Mirrored bus interfaces shall be looked up as if they were not mirrored.	No	Port maps are optional, even on buses with required ports. See also 6.20. The third possible presence value (**optional**) neither forces nor forbids the inclusion of the signal in the port map.
SCR 6.9		Only one component port in a port connection equivalence class may have the direction **out**.	No	
SCR 6.11		Only one component port in a port connection equivalence class may have the initiative **provides**.	No	
SCR 6.12		If abstraction definition `A` extends abstraction definition `B`, then abstraction definition `A` needs to have port elements for every port declared in abstraction definition `B`.	No	

Table B6—Ports (Continued)

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 6.13		If the abstraction definition referenced by a bus interface specifies a port is a wire port (i.e., the **port** element contains a **wire** sub-element), the port map shall only map that logical port to a wire component port.	No	
SCR 6.14		If the abstraction definition referenced by a bus interface specifies a port is a transactional port (i.e., the **port** element contains a **transactional** sub-element), the port map shall only map that logical port to a transactional component port.	No	
SCR 6.15		For any port connection equivalence class containing at least one physical **in** port, only one logical port of that port connection equivalence class shall be a port of a bus interface that has an interconnection to a bus interface using a different abstraction.	No	This rule prevents shared signals from crossing abstractions boundaries, since abstractors cannot describe the handling of such signals.
SCR 6.16		For any port connection equivalence class containing at least one physical **requires** port, only one logical port of that port connection equivalence class shall be a port of a bus interface that has an interconnection to a bus interface using a different abstraction.	No	This is the equivalent of rule 6.15 for transactional ports.
SCR 6.17		The value of the **group** sub-element of an **onSystem** element shall match the value of one of the system group names referenced in the bus definition referenced by the abstraction definition containing the **onSystem** element.	No	
SCR 6.18		The value of the **group** sub-element of a **system** element shall match the value of one of the system group names referenced in the bus definition referenced by the bus interface containing the **onSystem** element.	No	
SCR 6.19		If an abstraction definition's **busType** element references an addressable bus, the abstraction definition shall contain at least one **port** and **isAddress** sub-element.	No	
SCR 6.20		If the abstraction definition referenced by a bus interface specifies, for a port, a presence value of **illegal** for that interface mode of bus interface, and the bus interface has a port map, the port shall not be in that port map. For system interfaces, the port presence shall be looked up from the **onSystem** element with the group name matching that of the bus interfaces. Mirrored bus interfaces shall be looked up as if they were not mirrored.	No	Port maps are optional, even on buses with required ports. See also ~~6.20~~. The third possible presence value (**optional**) neither forces nor forbids the inclusion of the port in the port map.

Delete these definitions and/or move them into the appropriate Schema section

B.0.6 Port connection equivalence class

The *port connection equivalence class* of a (logical or component) port is the set of model and logical ports that can be reached from that port through any sequence of:

a) Bus interfaces' logical to physical port maps.

b) Interconnections between logical ports implied by interconnections between bus interfaces using the same abstraction of the bus.

c) Ad-hoc connections.

B.0.7 Addressable bus interface

A bus interface shall be *addressable* if its **isAddressable** element has the value *True*.

Table B7—Registers

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 7.1	50.	No register shall have an **addressOffset** that falls within the address range of another register in the same address block. The address range of a register is the half open range `[addressOffset, addressOffset + (size +bitsInLau -1) ÷ bitsInLau)`.	Yes	I.e., registers shall not overlap.
SCR 7.2	51.	No bit field shall have a **bitOffset** value that falls within the bit range of another bit field. The range of a bit field is the half open range `[bitOffset, bitOffset+width)`.	Yes	I.e., bit fields shall not overlap.
SCR 7.3	52.	Any register in an address block shall fall entirely within that address block. I.e., for every register `0 addressOffset addressBlockRange – registerSize`; where **addressBlockRange** is the **range** of the address block and **registerSize** is the size of the register in least addressable units (`((size +bitsInLau -1) ÷ bitsInLau)`.	Yes	
SCR 7.4	53.	Any bit field in a register shall fall entirely within that register. I.e., for every bit field `0 bitOffset RegisterSize – bitFieldWidth`; where **RegisterSize** is the **size** (in bits) of the register, and **bitFieldWidth** is the **width** of bit field.	Yes	

Table B8—Memory maps

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 8.1	54.	The width of an address block included in a memory map shall be a multiple of the memory map's **bitsInLau**.	Yes	
SCR 8.2	55.	Neither a parallel bank, nor banks within a parallel bank, shall contain subspace maps.	Yes	
SCR 8.3	56.	If a parallel bank contains a serial bank, the widths of all address blocks and sub-banks of that serial bank shall have identical widths.	Yes	I.e., the serial bank has a fixed, well-defined width. This is required for sensible addressing of the locations in a parallel bank.

Table B9—Addressing

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 9.1	57.	A non-hierarchical addressable master bus interface shall have an **addressSpaceRef** sub-element.	No	Since there are potentially useful applications of IP-XACT that do not require addressing information, failure to obey this rule should be treated as a warning rather than an error.
SCR 9.2	58.	A non-hierarchical addressable slave bus interface shall have a **memoryMapRef** sub-element or one or more **bridge** sub-elements referencing addressable master bus interfaces.	No	Since there are potentially useful applications of IP-XACT that do not require addressing information, failure to obey this rule should be treated as a warning rather than an error.
SCR 9.3		Only an address space referenced by the **addressSpaceRef** sub-element of a **cpu** element may contain an **exectutableImage** sub-element.		

Table B10—Hierarchy

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 10.1	59.	All members of a hierarchical family of bus interfaces shall reference the same **busDefinition** in their **busType** sub-elements	No	
SCR 10.2	60.	All members of a hierarchical family of bus interfaces shall have the same interface mode (**master**, **slave**, **system**, etc.)	No	
SCR 10.3	61	If any member of a hierarchical family of bus interfaces has a **connection** sub-element with a value other than **explicit** (the default), then all the sub-element values need to be identical.	No	
SCR 10.4	62	If any member of a hierarchical family of bus interfaces has an **index** sub-element, they all shall have identical **index** sub-elements.	No	
SCR 10.5	63.	If any member of a hierarchical family of bus interfaces has a **bitSteering** sub-element, they all shall have identical **bitSteering** sub-elements.	No	
SCR 10.6	64.	If any member of a hierarchical family of bus interfaces has a **portMap** sub-element, they all shall.	No	
SCR 10.7	65.	All the **portMap**s of a hierarchical family of bus interfaces reference the same set of bus ports, i.e., if one contains a port with the **busPortName** element and the value **s**, they all shall.	No	An effect of this, together with 9.1 and 9.2, is when a hierarchical bus interface is addressable, its non-hierarchical descendents (i.e., the leaves of the tree) also are, and, hence, they contain addressing information.
SCR 10.8	66.	In a hierarchical family of bus interfaces, all ports in the **portMap**s referencing the same bus port shall have the same **left** and **right** values.	No	
SCR 10.9	67.	In a hierarchical family of bus interfaces, the **componentPortName** of all ports in the **portMap** referencing the same bus port shall reference ports with the same direction.	No	
SCR 10.11	68.	In a hierarchical family of bus interfaces, if the component ports referenced by the **componentPortName** of all ports in the port maps referencing the same bus port have default values, they shall have identical default values.	No	I.e., it is legal for any descriptions of a port to have default values, but those that have default values shall have identical default values.

Table B10—Hierarchy (Continued)

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 10.12	69.	In a hierarchical family of bus interfaces, the **componentPortName** of all ports in the **portMap** referencing the same bus port shall reference ports with identical **clockDriver** sub-elements.	No	
SCR 10.13	70.	In a hierarchical family of bus interfaces, the **componentPortName** of all ports in the **portMap** referencing the same bus port shall reference ports with identical **singleShotDriver** sub-elements.	No	
SCR 10.14	71.	In a hierarchical family of bus interfaces, the **componentPortName** of all ports in the **portMap** referencing the same bus port shall reference ports with identical **portConstraintSets** sub-elements.	No	

Table B11—Hierarchy and memory maps

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 11.1	72.	In a hierarchical family of slave or mirrored-master bus interfaces, all bus interfaces that define addressing information shall define the same set of addresses to be visible.	No	I.e., if one member of the family defines an address as a valid address accessible through that bus interface, all members of the family that define addressing information shall define that same address as a valid address accessible through that bus interface.
SCR 11.2	73.	For any member of a hierarchical family of slave or mirrored-master bus interfaces, if an address resolves to reference a location outside the containing hierarchical family of components, that address shall reference the same location (i.e., the same address on the same bus) in every member of the hierarchical family that defines addressing information.	No	I.e., if `C` is a hierarchical component and the IP-XACT description of `C` itself or some design of `C` specifies accessing address `a` of `C` on bus interface `I` results in an access to address `b` of some other bus interface `J` of `C`, all designs of `C` that specify addressing on `I` shall indicate the same about this address.

Table B11—Hierarchy and memory maps (Continued)

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 11.3	74.	If any bit address (i.e., address plus bit offset) is resolved to a bit within an address block by any member of a hierarchical family of slave bus interfaces, all members of that family with addressing information shall resolve that bit address to a bit with identical behavioral properties.	No	If an address resolves to a location within the hierarchical family of components, its only observable features from outside the hierarchical family are its behavioral properties (except as defined in rule 11.4)
SCR 11.4	75.	When any two addresses resolve to the same location in the addressing information of any member of a hierarchical family of bus interfaces, this shall be true for all members of the hierarchical family of bus interfaces that have addressing information.	No	I.e., aliasing of addresses shall be preserved. Aliasing is observable from outside the hierarchical family.

Table B12—Constraints

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 14.1		A component wire port with direction **out** shall not have a drive constraint.	Yes	
SCR 14.2		A component wire port with a direction **in** shall not have a load constraint.	Yes	
SCR 14.3		An **onMaster**, **onSlave**, or **onSystem** element of a wire port with direction **out** shall not contain a drive constraint within its **modeConstraint** element.	Yes	
SCR 14.4		An **onMaster**, **onSlave**, or **onSystem** element of a wire port with direction **in** shall not contain a load constraint within its **modeConstraint** element.	Yes	
SCR 14.5		An **onMaster**, **onSlave**, or **onSystem** element of a wire port with direction **out** shall not contain a load constraint within its **mirroredModeConstraint** element.	Yes	
SCR 14.6		An **onMaster**, **onSlave**, or **onSystem** element of a wire port with direction **in** shall not contain a drive constraint with its **mirroredModeConstraint** element.	Yes	

Table B12—Constraints (Continued)

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 14.7		The **clockName** in a timing constraint of a component port shall be the name of another component port of the component or an **otherClockDriver** of the component.	Yes	
SCR 14.9		The **clockName** in a timing constraint of a port within an abstraction definition shall be the name of another port of the abstraction definition; that referenced port shall have an **isClock** sub-element.		

Table B13—Design configurations

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 15.1		The value of any **generatorName** element shall match the value of a **name** sub-element of a **generator** element in the generator chain referenced by the **generatorChain** element enclosing the **generatorName** element.	No	
SCR 15.2		The value of an **instanceName** within a **viewConfiguration** shall match the value of the **instanceName** element of a **componentInstance** of the design document referenced by the design configuration document containing the **viewConfiguration** element.	No	
SCR 15.3		The value of an **viewName** within a **viewConfiguration** shall match the value of the **name** element of a view within the component referenced by the component instance that is itself referenced by the **instanceName** sub-element of the **viewConfiguration** element.	No	

Table B14—Rules requiring external knowledge

Rule number	V1.2 rule number	Rule	Single doc check	Notes
SCR 12.1	76.	The **name** sub-element of a **file** element can contain environment variables in the form of `${ENV_VAR}` which are meaningful to the host operating system and, when expanded, shall result in a string which is a valid URI.	Yes	
SCR 12.2	77.	In VLNVs, the vendor name shall be specified as the top-level internet domain name for that organization. The domain shall be ordered with the top-level domain name at the end (as in HTTP URLs), e.g., `mentor.com`, `arm.com`, etc.	Yes	This is to guarantee uniqueness of vendor names.
SCR 12.3	78.	The **envIdentifier** of a **view** shall be a text string consisting of three fields delimited by colons (**:**). The first two fields shall be a language name, which shall be one of the languages available for **fileTypes**, and a tool name. The tool name may be generic (e.g., `*Simulation` or `*Synthesis`) or a specific tool name, such as `DesignCompiler` or `VCS`. The third field shall be an arbitrary vendor-specific text string.	Yes	Tool vendors need to publish a list of valid tool names in the SPIRIT web site.

Annex C

(normative)

Types

Many elements and attributes defined in the standard have associated types. These types define the legal values and ranges for input into these element and attributes.

C.1 boolean

The **boolean** type defines two possible value, ***True*** and ***False***.

C.2 configurableDouble

The **configurableDouble** type defines a decimal floating point number of IEEE### precision, containing the numbers `0-9`.

C.3 float

The **float** type defines a decimal floating point number of IEEE### precision, containing the numbers `0-9`.

C.4 integer

The **integer** type defines a decimal integer number of infinite precision, containing the numbers `0-9`.

C.5 Name

The **Name** type defines a series of any characters, excluding whitespace characters.

C.6 NMTOKEN

The **NMTOKEN** type defines a series of any characters, excluding whitespace characters.

C.7 nonNegativeInteger

The **nonNegativeInteger** type is a subtype of **integer**; it follows all the same rules, except its value shall be greater than or equal to `0`.

C.8 positiveInteger

The **positiveInteger** type is a subtype of **integer**; it follows all the same rules, except its value shall be greater than `0`.

C.9 scaledInteger

The **scaledInteger** type defines an integer of infinite precision. The number may be in any of the follow formats with or without a leading +/- indication.

a) Decimal containing numbers `0-9`.

b) Hexadecimal representation starting with `0x` or `#`, and containing the numbers `0-9` and letters `A-F` (case-insensitive).

c) Optionally, the number may end with the following case-insensitive suffixes. Each suffix is a multiplier of the resulting value.

   1) `K` is a multiplier of 1024.

   2) `M` is a multiplier of 1024*1024.

   3) `G` is a multiplier of 1024*1024*1024.

   4) `T` is a multiplier of 1024*1024*1024*1024.

Example: `4K` evaluates to `4096`. `0x1000` evaluates to `4096`.

C.10 scaledNonNegativeInteger

The **scaledNonNegativeInteger** type is a subtype of **scaledInteger**; it follows all the same rules, except its value shall be greater than or equal to `0`.

C.11 scaledPositiveInteger

The **scaledPositiveInteger** type is a subtype of **scaledInteger**; it follows all the same rules, except its value shall be greater than `0`.

C.12 SpiritURI

The **SpiritURI** type defines a path to a file, directory, or executable in URI format. **Any additional constraints??

C.13 string

The **string** type defines a series of any characters and may include spaces.

Annex D

(normative)

Dependency XPATH

This version of the standard utilizes XPATH 1.0 as a means to specify an equation for the contents of a resolvable element. This is done by setting the **resolve** attribute to `resolve="dependent"`. When the resolve attribute is set to dependent the dependency attribute is required.

The accuracy of the XPATH functions if numeric shall be of infinite precision and not limited to any fixed number of bits. This is necessary to ensure that all systems are interoperable and that the large calculations required by configuration of IP-XACT components is successful.

In addition to the standard XPATH 1.0 functions (add xref), IP-XACT defines the following four extra functions to aid expressions calculations.

D.1 spirit:containsToken

```
spirit:containsToken(string, string)
```

The **containsToken** function (Boolean) returns *True* if the first argument string contains the second argument string as a token and otherwise returns *False*. To be interpreted as a token, the second string needs to be found within the first string and be separated by white space from any other characters in the first string that are not white space characters.

Purpose: Some attributes in IP-XACT are a list of tokens separated by white space. This function allows XPATH selection based on whether the attribute contains a specific token.

Example: `spirit:containsToken('default spine driver','pin')` evaluates to *False*, whereas the standard XPATH function **contains** would evaluate to *True* with the same arguments.

D.2 spirit:decode

```
spirit:decode(string)
```

The **decode** function (number) decodes the string argument to a number and returns the number or NaN (if the string cannot be decoded). If the argument is omitted, it defaults to the context node converted to a string. If the string argument is a decimal formatted number, it is returned unchanged. If it is a hexadecimal representation starting with `0x` or `#`, it is converted to a decimal number and returned. If it is in engineering notation ending in a `k`, `m`, `g`, or `t` suffix (case-insensitive), the numeric part is multiplied by the appropriate power of two. `K` is a multiplier of 1024. `G` is a multiplier of 1024*1024. `G` is a multiplier of 1024*1024*1024. `T` would equal a multiplier of 1024*1024*1024*1024.

Purpose: IP-XACT allows numbers to be expressed in hexadecimal format and engineering format. When setting up dependencies on configurable values, it is sometimes necessary to perform some arithmetic in the dependency XPATH expression. However, XPATH only supports arithmetic on numbers and it only recognizes decimal strings as numbers. This function allows the alternate formats to be converted to numbers recognizable by XPATH.

Example: `spirit:decode('0x4000')` evaluates to 16384. `spirit:decode('4G')` evaluates to 4294967296.

D.3 spirit:pow

`spirit:pow(`*`number, number`*`)`

The **pow** function (number) returns a number, which is the first argument raised to the power of the second argument.

Purpose: It is common for a component to have a configurable number of address bits. When this happens, the size of the address range it occupies on a memory map varies exponentially with the number of address bits. This function gives XPATH the mathematical capabilities needed to describe this relationship in a dependency expression.

Example: `spirit:pow(2, 10)` evaluates to 1024.

D.4 spirit:log

`spirit:log(`*`number, number`*`)`

The **log** function (number) returns a number that is the log of the second argument in the base of the first argument.

Purpose: This is the inverse of **pow** function. It is intended to express the reverse of the dependency described for the **pow** function. In this case, the range of an address block might be configurable and the number of address bits might be expressed as a dependency of the address range using the log function.

Example: `spirit:log(2, 1024)` evaluates to 10.

D.5 Example

```
<spirit:memoryMaps>
  <spirit:memoryMap>
    <spirit:name>mmap</spirit:name>
    <spirit:addressBlock>
       <spirit:name>ab1</spirit:name>
      <spirit:baseAddress  spirit:resolve="user" spirit:id="baseAddress">0</
   spirit:baseAddress>
      <spirit:bitOffset>0</spirit:bitOffset>
      <spirit:range spirit:id="range">786432</spirit:range>
      <spirit:width>32</spirit:width>
      <spirit:usage>memory</spirit:usage>
      <spirit:access>read-write</spirit:access>
    </spirit:addressBlock>
  </spirit:memoryMap>

  <spirit:memoryMap>
    <spirit:name>dependent_mmap</spirit:name>
    <spirit:addressBlock>
```

```
<!-- The baseAddress in this memoryMap is dependent on the previous memory map
    and the formula to compute the baseAddress from the baseAddress of previous
    map is expressed as an XPATH expression -->

      <spirit:baseAddress  spirit:resolve="dependent"
    spirit:dependency="spirit:pow(2,floor(spirit:log(2,
    spirit:decode(id('baseAddress'))+ spirit:decode(id('range')))+1))"
    spirit:id="dependentBaseAddress">0</spirit:baseAddress>
      <spirit:bitOffset>0</spirit:bitOffset>
      <spirit:range>4096</spirit:range>
      <spirit:width>32</spirit:width>
      <spirit:usage>register</spirit:usage>
      <spirit:access>read-write</spirit:access>
    </spirit:addressBlock>
  </spirit:memoryMap>
```

Annex E

(informative)

External bus vs. an internal/digital interface

While the current use of IP-XACT schema may be viewed as describing single chip implementations, the schemas works equally well at the package- and board-level. Often a `PHY` component exists which interconnects the internal and external bus. Some standards define both of these interfaces, some define only the internal, and some define only the external. A common point of confusion is to use an external bus standard as an interface on an internal component. This is legal if the component caries the full `PHY` implementation, but this often makes the component very technology- or implementation-dependant.

E.1 Example: ethernet interfaces

An Ethernet bus might be described as more than a single wire and in a system that includes Ethernet buses, it might also include all the interfaces shown in Figure E.1.

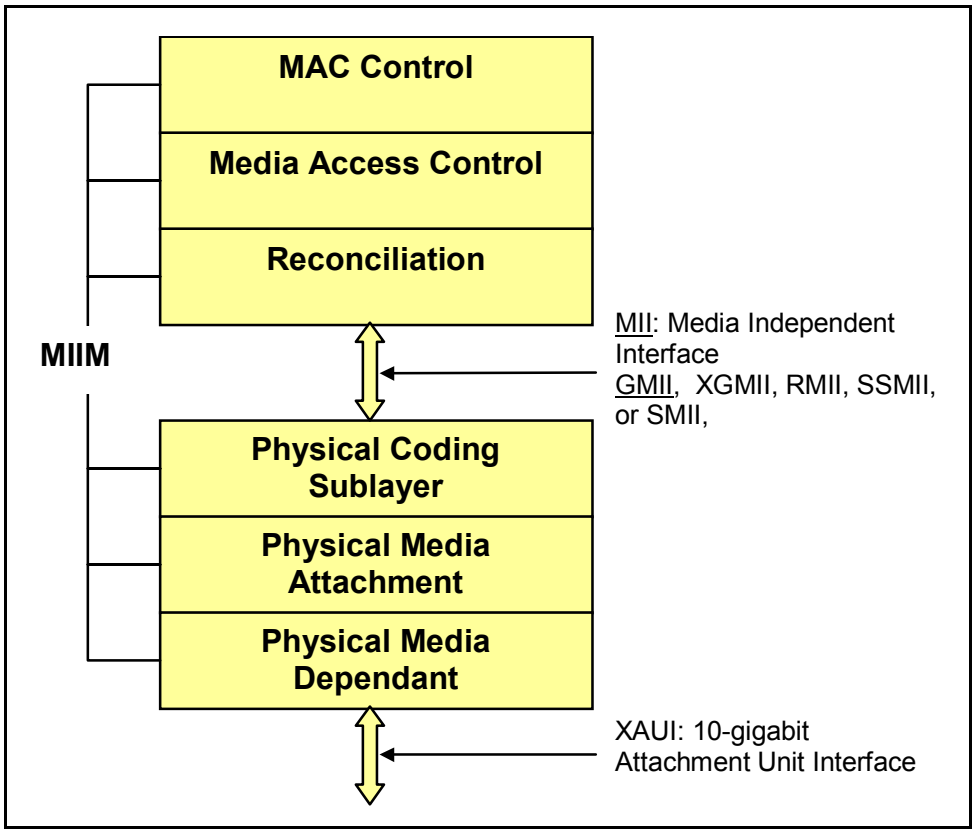


Figure E.1—Ethernet interface examples

XAUI: 10-gigabit Attachment Unit Interface
MII: Media Independent Interface
GMII: Gigabit Media Independent Interface

XGMII: 10-gigabit media-independent interface
RMII: Reduced MII, 7-pin interface
SSMII: Source Synchronous MII
SMII: Serial Media Independent Interface, this provides an interface to Ethernet MAC. The SMII provides the same interface as the MII, but with a reduced pinout. The reduction in ports is achieved by multiplexing data and control information to a port transmit port and a single receive port.

E.2 Example: I²C bus

The I2C eye-squared-see bus is a two-wire bus with a clock and data line. The standard described bus is the two-wire bus. IP-XACT has defined an additional, related bus that is the internal digital interface. The reference `BusSpec` shown in Figure E.2 contains three pins for each external pin: for `SDA` (the data line), the internal pins are defined as input, output, and enable as `SDA_I`, `SDA_O`, and `SDA_E`; in a similar manner, for the clock bus `SCL`, the internal pins are defined again for the functions of input, output, and enable as `SCL_I`, `SCL_O`, and `SCL_E`.

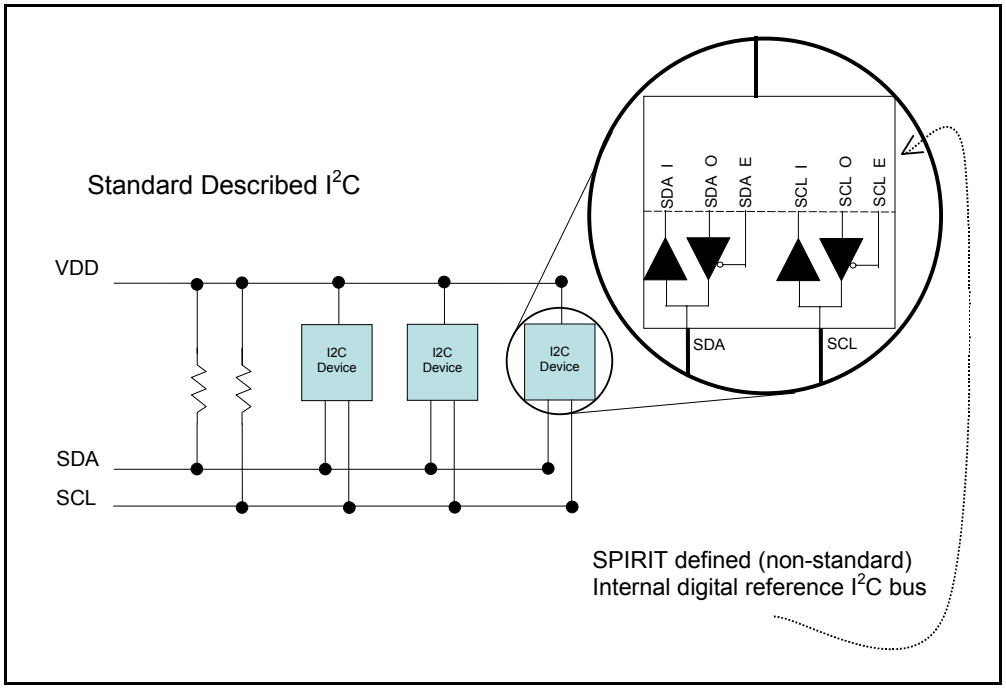


Figure E.2—I2C interface example