IEC 62014-4

Edition 1.0 2015-03

INTERNATIONAL STANDARD

IEEE Std 1685™-2009

IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows

IEC 62014-4:2015-03(en) IEEE Std 1685-2009

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland
Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00
info@iec.ch
www.iec.ch

Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue
New York, NY 10016-5997
United States of America
stds.info@ieee.org
www.ieee.org

IEC 62014-4

Edition 1.0 2015-03

INTERNATIONAL STANDARD

IEEE Std 1685™-2009

IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

ICS 25.040

ISBN 978-2-8322-2265-2

Warning! Make sure that you obtained this publication from an authorized distributor.

Contents

IP-XACT, STANDARD STRUCTURE FOR PACKAGING, INTEGRATING, AND REUSING IP WITHIN TOOL FLOWS

FOREWORD

1) The International Electrotechnical Commission (IEC) is a worldwide organization for standardization comprising all national electrotechnical committees (IEC National Committees). The object of IEC is to promote international co-operation on all questions concerning standardization in the electrical and electronic fields. To this end and in addition to other activities, IEC publishes International Standards, Technical Specifications, Technical Reports, Publicly Available Specifications (PAS) and Guides (hereafter referred to as "IEC Publication(s)"). Their preparation is entrusted to technical committees; any IEC National Committee interested in the subject dealt with may participate in this preparatory work. International, governmental and non-governmental organizations liaising with the IEC also participate in this preparation.

   IEEE Standards documents are developed within IEEE Societies and Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. IEEE develops its standards through a consensus development process, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of IEEE and serve without compensation. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards. Use of IEEE Standards documents is wholly voluntary. IEEE documents are made available for use subject to important notices and legal disclaimers (see http://standards.ieee.org/IPR/disclaimers.html for more information).

   IEC collaborates closely with IEEE in accordance with conditions determined by agreement between the two organizations.

2) The formal decisions of IEC on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC National Committees. The formal decisions of IEEE on technical matters, once consensus within IEEE Societies and Standards Coordinating Committees has been reached, is determined by a balanced ballot of materially interested parties who indicate interest in reviewing the proposed standard. Final approval of the IEEE standards document is given by the IEEE Standards Association (IEEE-SA) Standards Board.

3) IEC/IEEE Publications have the form of recommendations for international use and are accepted by IEC National Committees/IEEE Societies in that sense. While all reasonable efforts are made to ensure that the technical content of IEC/IEEE Publications is accurate, IEC or IEEE cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.

4) In order to promote international uniformity, IEC National Committees undertake to apply IEC Publications (including IEC/IEEE Publications) transparently to the maximum extent possible in their national and regional publications. Any divergence between any IEC/IEEE Publication and the corresponding national or regional publication shall be clearly indicated in the latter.

5) IEC and IEEE do not provide any attestation of conformity. Independent certification bodies provide conformity assessment services and, in some areas, access to IEC marks of conformity. IEC and IEEE are not responsible for any services carried out by independent certification bodies.

6) All users should ensure that they have the latest edition of this publication.

7) No liability shall attach to IEC or IEEE or their directors, employees, servants or agents including individual experts and members of technical committees and IEC National Committees, or volunteers of IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board, for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication, use of, or reliance upon, this IEC/IEEE Publication or any other IEC or IEEE Publications.

8) Attention is drawn to the normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.

9) Attention is drawn to the possibility that implementation of this IEC/IEEE Publication may require use of material 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. IEC or IEEE shall not be held responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patent Claims or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility.

International Standard IEC 62014-4/ IEEE Std 1685-2009 has been processed through IEC technical committee 91: Electronics assembly technology, under the IEC/IEEE Dual Logo Agreement.

The text of this standard is based on the following documents:

IEEE Std	FDIS	Report on voting
1685 (2009)	91/1207/FDIS	91/1226/RVD

Full information on the voting for the approval of this standard can be found in the report on voting indicated in the above table.

The IEC Technical Committee and IEEE Technical Committee have decided that the contents of this publication will remain unchanged until the stability date indicated on the IEC web site under "http://webstore.iec.ch" in the data related to the specific publication. At this date, the publication will be

- reconfirmed,
- withdrawn,
- replaced by a revised edition, or
- amended.

IEEE Std 1685™-2009

IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows

Sponsor
Design Automation Standards Committee
of the
IEEE Computer Society
and the
IEEE Standards Association Corporate Advisory Group

Approved 9 December 2009
IEEE SA-Standards Board

Grateful acknowledgment is made to The SPIRIT Consortium, Inc., for permission to use the following source material:

IP-XACT 1.2 and IP-XACT 1.5

Abstract: Conformance checks for eXtensible Markup Language (XML) data designed to describe electronic systems are formulated by this standard. The meta-data forms that 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. A set of XML schemas of the form described by the World Wide Web Consortium (W3C®) and a set of semantic consistency rules (SCRs) are included. A generator interface that is portable across tool environments is provided. The specified combination of methodology-independent meta-data and the tool-independent mechanism for accessing that data provides for portability of design data, design methodologies, and environment implementations.
Keywords: abstraction definitions, address space specification, bus definitions, design environment, EDA, electronic design automation, electronic system level, ESL, implementation constraints, IP-XACT, register transfer level, RTL, SCRs, semantic consistency rules, TGI, tight generator interface, tool and data interoperability, use models, XML design meta-data, XML schema

IEEE Introduction

This introduction is not part of IEEE Std 1685-2009, IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows.

The purpose of this standard is to provide the electronic design automation (EDA), semiconductor, electronic design intellectual property (IP) provider, and system design communities with a well-defined and unified specification for the meta-data that 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 system on chip (SoC) design environments (DEs); 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. The data specified by IP-XACT is extensible in locations specified in the schema.

IP-XACT enables the use of a unified structure for the meta specification of a design, components, interfaces, documentation, and interconnection of components. This structure can be used as the basis of both manual and 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 that 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 eXtensible Markup Language (XML) bus and abstraction definitions in their set of deliverables. 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 SPIRIT Consortium Web site.[a]

[a]Available at http://www.spiritconsortium.org.

Notice to users

Laws and regulations

Users of these documents should consult all applicable laws and regulations. Compliance with the provisions of this standard does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

Updating of IEEE documents

Users of IEEE standards should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect. In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE Standards Association website at http://ieeexplore.ieee.org/xpl/standards.jsp, or contact the IEEE at the address listed previously.

For more information about the IEEE Standards Association or the IEEE standards development process, visit the IEEE-SA website at http://standards.ieee.org.

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: http://standards.ieee.org/reading/ieee/updates/errata/index.html. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations can be accessed at the following URL: http://standards.ieee.org/reading/ieee/interp/index.html.

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. The IEEE shall not be responsible for identifying patents or patent applications for which a license may be required to implement an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows

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) schema[1] for meta-data documenting *intellectual property* (IP) used in the development, implementation, and verification of electronic systems and an *application programming interface* (API) to provide tool access to the meta-data. This schema provides a standard method to document IP that is compatible with automated integration techniques.The API provides a standard method for linking tools into a *system development* framework, enabling a more flexible, optimized development environment. Tools compliant with this standard will be able to interpret, configure, integrate, and manipulate IP blocks that comply with the IP meta-data description. The standard is based on version 1.4 IP-XACT of The SPIRIT Consortium. The standard is independent of any specific design processes. It does not cover those behavioral characteristics of the IP that are not relevant to integration.

[1]Information on references can be found in Clause 2.

1.2 Purpose

This standard enables the creation and exchange of IP in a highly automated design environment.

1.3 Design environment

The IP-XACT specification is a mechanism to express and exchange information about design IP and its required configuration.[2] While the IP-XACT description 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 meta-data descriptions of the system on chip (SoC) such that its system design and implementation flows are efficiently enabled and reuse centric.

The use of the IP-XACT specified formats and interfaces are shown, in **bold**, in Figure 1 and described in the following subclauses.

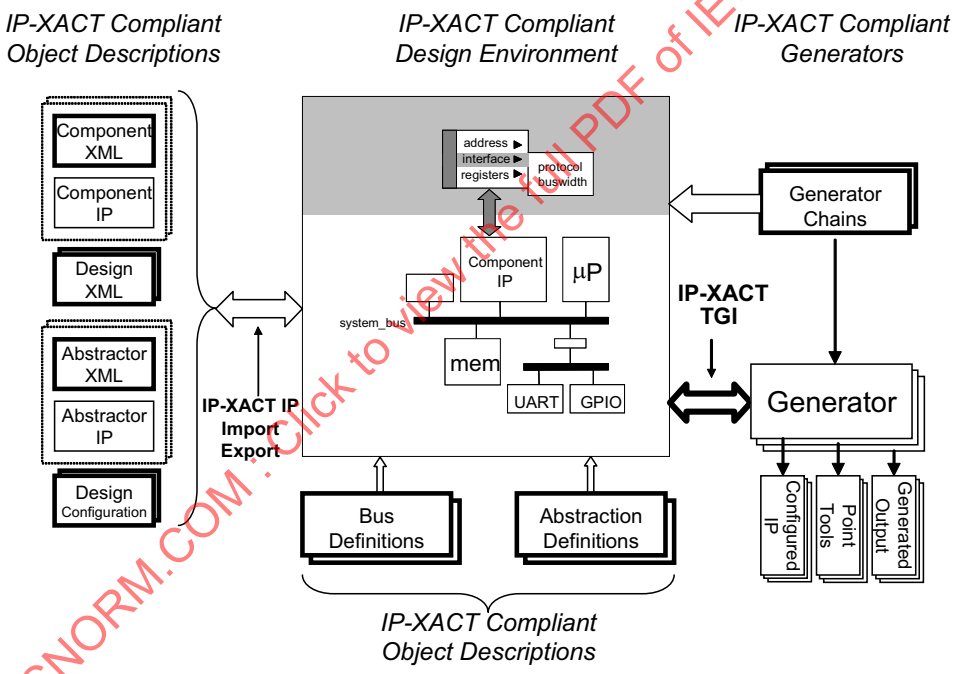


Figure 1—IP-XACT design environment

1.3.1 IP-XACT design environment

A DE enables 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 may provide two basic types of services: *design capture*, which is the expression of design configuration by the IP

[2]IP-XACT uses the World Wide Web Consortium (W3C®) standard for the XML version 1.0 data (http://www.w3.org/TR/2000/REC-xml-20001006). The valid format of that XML data is described in a *schema* by using the Schema Description Language described therein. W3C is a registered trademark of the World Wide Web Consortium.

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 shall 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 hardware description language (HDL) code] as well as the structure of the IP package (e.g., where design descriptions 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) that may be expressed as configurable parameters in the IP-XACT meta-data.

As part of design build, generators may be provided internally by 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 v1.5 enabled, a system design tool shall support the import/export of IP expressed with valid IP-XACT v1.5 meta-data for both component IP and designs, and it needs to support the tight generator interface (TGI) for interfacing with external generators (to the DE).

1.3.2 IP-XACT object descriptions

The IP-XACT schema is the core of the IP-XACT specification. There are seven top-level schema definitions. Each schema definition can be used to create object descriptions of the corresponding type.

- A *bus definition* description defines the type attributes of an bus.
- An *abstraction definition* description defines the representation attributes of a bus.
- A *component* description defines an IP or interconnect structure.
- A *design* description defines the configuration of and interconnection between components.
- An *abstractor* description defines an adaptor between interfaces of two different abstractions.
- A *generator chain* description defines the grouping and ordering of generators.
- A *design configuration* description defines additional configuration information for a generator chain or design description.

1.3.3 Object interactions

An object description contains a unique identifier in the header. The identifier in IP-XACT terms is called a VLNV after the four elements that define its value: vendor, library, name, and version. See C.6 for further details on a VLNV. This VLNV is used to create a reference from one description to another. The links between these objects are illustrated in Figure 2. The arrows (A → B) illustrate a reference of one object to another (e.g., reference of object `B` from object `A`).

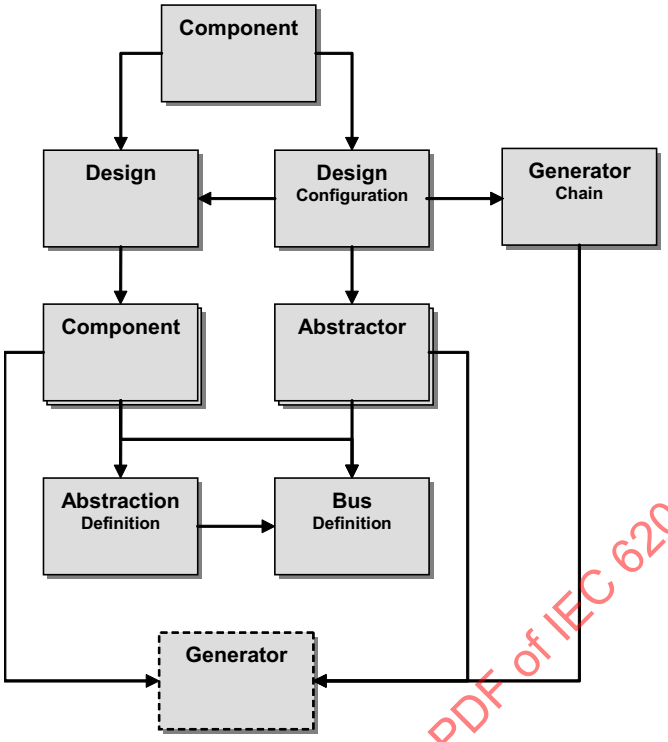


Figure 2—IP-XACT object interactions

1.3.4 IP-XACT generators

Generators are executable objects (e.g., scripts or binary programs) that may be integrated within a DE (referred to as *internal*) or provided separately as an executable (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 DE (e.g., an external design netlister, external design checker).

An internal generator may perform a wide variety of tasks and may access IP-XACT compliant meta-data by any method a DE supports. IP-XACT does not describe these protocols.

An *external generator* (often referred to as a *TGI generator*) is an executable program or script invoked from within a DE to query or configure design descriptions and their related component and abstractor descriptions. External generators can use the TGI to *access* their IP-XACT meta-data descriptions (as currently loaded into the DE) and perform the various operations associated with those descriptions. In addition, external generators shall only *operate* upon IP-XACT compliant meta-data through the defined TGI, see 1.3.6.

Generators can be referenced from a component, abstractor, or generator chain description. Generators can also be grouped and ordered in generator chain descriptions and those chain descriptions contained inside other chain descriptions. This sequencing of generators is *critical* for providing script-based support for SoC flow creation.

1.3.5 IP-XACT design environment interfaces

There are two obvious interfaces expressed in Figure 1: from the DE to the external IP libraries and from the DE to the generators. In the former case, the IP-XACT specifications are *neutral* regarding the design tool

interfaces to IP repositories. Being able to read and write IP with IP-XACT meta-data is required; however, the *formal interaction* between an external IP repository and a DE is not specified.

1.3.6 Tight generator interface

The *tight generator interface* (TGI) is the method a generator uses to efficiently access a design or component description in a DE-independent and generator-language-independent manner. Therefore, a generator running on two different DEs produces the same results. The DE and the generator communicate with each other by sending messages utilizing the Simple Object Access Protocol (SOAP) standard version 1.2[3] specified in the Web Services Description Language (WSDL) version 1.1.[4] SOAP provides a simple means for sending XML-format messages using the Hypertext Transfer Protocol (HTTP) or other transport protocols. IP-XACT supports using an HTTP protocol or a file protocol.

The SOAP messages passed between the generator and the DE allow the generator to get all information about the design interconnections (which contain components and abstractors), provide set information for any configurable elements in a component or abstractor, and make simple modifications of the design description. For additional details on the DE generator invocation and the SOAP messages passed between the generator and the DE, see Annex G.

1.3.7 Design intellectual property

IP-XACT is structured around the concept of IP reuse. *Electronic design intellectual property*, or IP, is a term used in the ED community to refer to a reusable collection of design specifications that represent the behavior, properties, and/or representation of the design in various media. The name IP is partially derived from the common practice of considering a collection of this type to be the intellectual property of one party. Both hardware and software collections are encompassed by this term.

These collections may include the following:

a) Design objects—This can include the following:
   1) Transaction-level modeling (TLM) descriptions: SystemC® and SystemVerilog [5]
   2) Fixed HDL descriptions: Verilog®, VHDL[6]
   3) Configurable HDL descriptions (e.g., bus-fabric generators)
   4) Design models for register transfer level (RTL) and transactional simulation (e.g., compiled core models)
   5) HDL-specified verification IP (VIP) (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 v1.5, 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: standard, user guide, etc.

IP-XACT XML meta-data descriptions provide a standardized way of collecting much of the structural information contained in the file sets. IP-XACT also can contain the information that identifies the appropriate files included in a collection to be used for different parts of the design process.

[3]Available from the W3C Web site at http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

[4]Available from the W3C Web site at http://www.w3.org/TR/wsdl.

[5]SystemC is a registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries. Verilog is a registered trademark of Cadence Design Systems, Inc. in the United States and/or other jurisdictions. This information is given for the convenience of users of this standard and does not constitute an endorsement by the IEEE of these products. Equivalent products may be used if they can be shown to lead to the same results.

[6]See Footnote 5.

1.4 IP-XACT Enabled implementations

Complying with the rules outlined in this subclause 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 subclause 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 descriptions.

   2) Parsing completeness: Cannot require information that could be expressed in an IP-XACT format to be specified in a non-IP-XACT format. Processing of all information present in an IP-XACT 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 (SCRs) 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 subclauses describe the rules a provider has to check to claim a product is IP-XACT Enabled.

An IP-XACT Enabled DE or point tool may read descriptions based on multiple versions of the IP-XACT schema. If the DE or point tool does provide this capability, the effect shall be as if all of the descriptions had been translated by the XSL Transform (XSLT) version 1.0,[7] which is provided with the IP-XACT release to convert descriptions from one version to the next. In addition, a DE or point tool may preserve information in the initial description for use outside of the scope of the IP-XACT specification.

1.4.1 Design environments

An IP-XACT Enabled DE:

a) Shall follow the parse validity requirements shown in 1.4.

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

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

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.

XPATH version 1.0[8] support is required for DE-compliance.

[7]Available from the W3C Web site at http://www.w3.org/TR/1999/REC-xslt-19991116.
[8]Available from the W3C Web site at http://www.w3.org/TR/1999/REC-xpath-19991116.

1.4.2 Point tools

A point tool is a tool that has a particular rather than a general set of capabilities. In contrast to IP-XACT Enabled DE (see 1.4.1), an IP-XACT Enabled point tool:

a) Shall follow the parse validity requirements shown in 1.4.

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

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

1.4.3 IPs

An IP-XACT Enabled IP:

a) Shall have an IP-XACT description that follows the description and semantic validity requirements shown in 1.4.

b) Shall only use IP-XACT Enabled generators for any generators associated with this IP.

XML descriptions compliant to IP-XACT shall provide a namespace reference to the `index.xsd` schema file, not to any of the other files in the release.

1.4.4 Generators

An IP-XACT Enabled generator:

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

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

c) Shall be callable though the IP-XACT TGI (see Annex G).

d) Shall only communicate with the DE that invoked it through the IP-XACT TGI (see Annex G).

1.5 Conventions used

The conventions used throughout the document are included here.

IP-XACT is case-sensitive.

1.5.1 Visual cues (meta-syntax)

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

Bold italics shows group names or data types, e.g., ***nameGroup*** or ***boolean***. For definitions of types, see Annex D.

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

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 [B5].[9]

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

The W3C Web site[10] specifies the XML schema language used to define the IP-XACT XML schemas. Normative details for compliance to the IP-XACT standard are 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 the schema files is intended to be identical. Where the figures and schema are in conflict, the XML schema file shall take precedence.[11]

1.5.4.1 Elements and attributes

The *element* is the fundamental building block on which this standard is based. An element may be either a *leaf element*, which is a container for information, or a *branch element*, which may contain further branch elements or leaf elements.

A leaf or branch element may also contain *attributes*. Attributes are containers for information within the containing element.

1.5.4.2 Types

A *type* is a designation of the format for the contents of an element or attribute. There are two different styles of types that can be defined. A type may be assigned to a leaf element or an attribute. This type is called a ***simpleType*** and defines the format of data that may be stored in this container. A type may also be assigned to a branch element. This type is called a ***complexType*** and defines further elements and attributes contained in the branch element.

1.5.4.3 Groups

A group is a collection of elements or attributes, which allow the same collection of items to be referenced consistently in many places. There are two different types of groups that can be defined. A *group* is a combination of leaf or branch elements; an ***attributeGroup***, a simple list of attributes. The names assigned to either group have no representation in the resulting description.

1.5.4.4 Namespace

Each element, attribute, type, or group has a name, which is preceded by a namespace and separated from the name by a colon (**:**). For the examples in 1.5.4.5, **xyz** is used as the namespace for all of the items

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

[10]Available from the W3C Web site at http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

[11]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®. XMLSpy is a registered trademark of Altova GmbH. This information is given for the convenience of users of this standard and does not constitute an endorsement by the IEEE of this product. Equivalent products may be used if they can be shown to lead to the same results.

whereas the standard uses **spirit**. Within the text of this standard, the namespace is not written when describing an item; it is only shown in examples.

1.5.4.5 Diagrams

The diagrams used throughout this standard graphically detail the organization the elements and attributes.

1.5.4.5.1 Elements and sequences

Figure 3 shows the sequence-compositor. At the left is a branch element, **element1**, with some descriptive text below. **element1** is connected to a sequence-compositor. The sequence-compositor defines the order the subelements appear in the branch element. **subElement1** shall appear first inside of **element1**. This is followed by **subElement2**, **subElement3**, **subElement4**, and **subElement5** before closing **element1**.

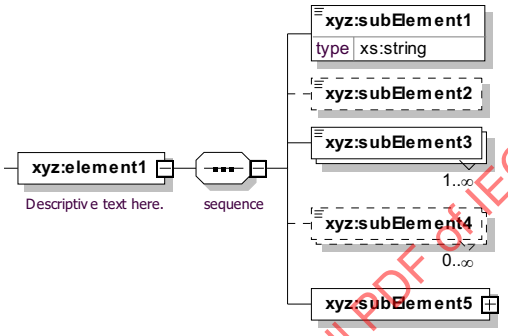


Figure 3—Sequence-compositor

a) **subElement1** is a mandatory element, as indicated by the solid line of the containing box. The type of the data contained in this element is set to ***string*** and it has a default value of `ip-xact` if the element is present, but left empty.

b) **subElement2** is an optional element, as indicated by the dashed-line of the containing box.

c) **subElement3** is an mandatory element that may appear multiple times, indicated by the double-solid line of the containing box. The number of times the element may appear is indicated by the range of the numbers listed below the element.

d) **subElement4** is an optional element that may appear multiple times, as indicated by the double-dashed line of the containing box. The number of times the element may appear is indicated by the range of the numbers listed below the element.

e) **subElement5** is an mandatory branch element that contains further elements inside, as indicated by the small plus sign (+) in the small box on the right.

Figure 4 shows variations of a sequence-compositor. **root1** is connected to an optional sequence-compositor, as indicated by the symbol being drawn with a dashed line. **element1** may appear first inside of **root1**; if it does, it shall be followed by **element2**. Each subelement is connected to a sequence-compositor.

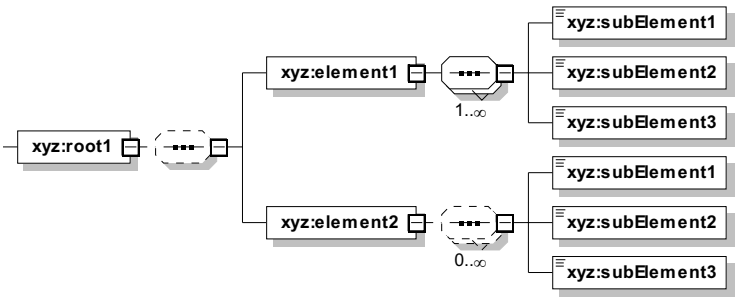


Figure 4—Sequence-compositor variations

— **element1** may contain one or more of the following sequences in the following order: **subElement1** and **subElement2** and **subElement3**. The number of times the sequence-compositor may appear is indicated by the range of the numbers listed below the symbol. If the range is greater than 1, the sequence-compositor symbol is drawn with double lines.

— **element2** is optional and may contain one or more of the following sequences in the following order: **subElement1** and **subElement2** and **subElement3**. The number of times the sequence-compositor may appear is indicated by the range of the numbers listed below the symbol. If the range starts at 0 and the maximum is greater then 1, the sequence-compositor symbol is drawn with double-dashed lines.

1.5.4.5.2 Elements and choices

Figure 5 shows the variations of the choice-compositor. **root** is connected to a choice-compositor. The choice-compositor specifies that one of the elements on the right side shall be chosen. **root** may contain one of the following: **element1**, **element2**, or **element3**. Each subelement is connected to a choice-compositor.

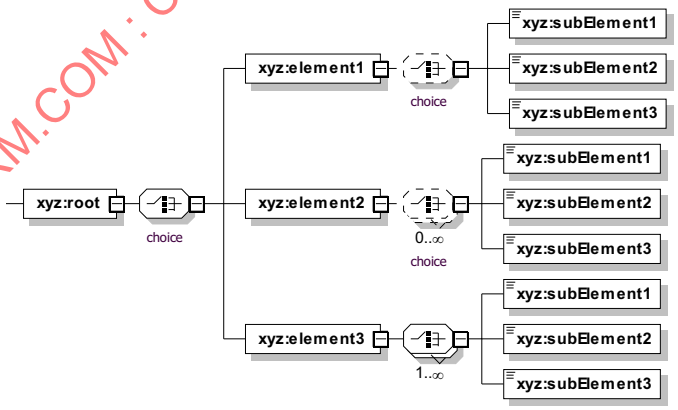


Figure 5—Choice-compositor variations

a) **element1** may contain one of the following: **subElement1**, **subElement2**, or **subElement3**, as indicated by the symbol being drawn with a dashed line.

b) **element2** may contain any (0 or more) of the following: **subElement1**, **subElement2**, or **subElement3** in any order. The number of times the choice-compositor may appear is indicated by

the range of the numbers listed below the symbol. If the range starts at 0, the choice-compositor is drawn with dashed lines.

c) **element3** may contain one or more of the following: **subElement1**, **subElement2**, or **subElement3** in any order. The number of times the choice-compositor may appear is indicated by the range of the numbers listed below the symbol. If the range is greater than 1, the choice-compositor is drawn with double lines.

1.5.4.5.3 Elements, attributes, groups, and attributeGroups

Figure 6 shows the use of attributes, groups, and attributeGroups. **element1** contains two attributes, shown in the tab shaped box labeled *attributes*. **attribute1** is optional, as indicated by the dashed containing box. **attribute1** also has a type defined of ***integer*** and a default value of 7 if the attribute is not present. **attribute2** is a required attribute, as indicated by the solid containing box, and is of type ***boolean*** with no default. The ordering in which **attribute1** and **attribute2** appear inside **element1** is irrelevant.

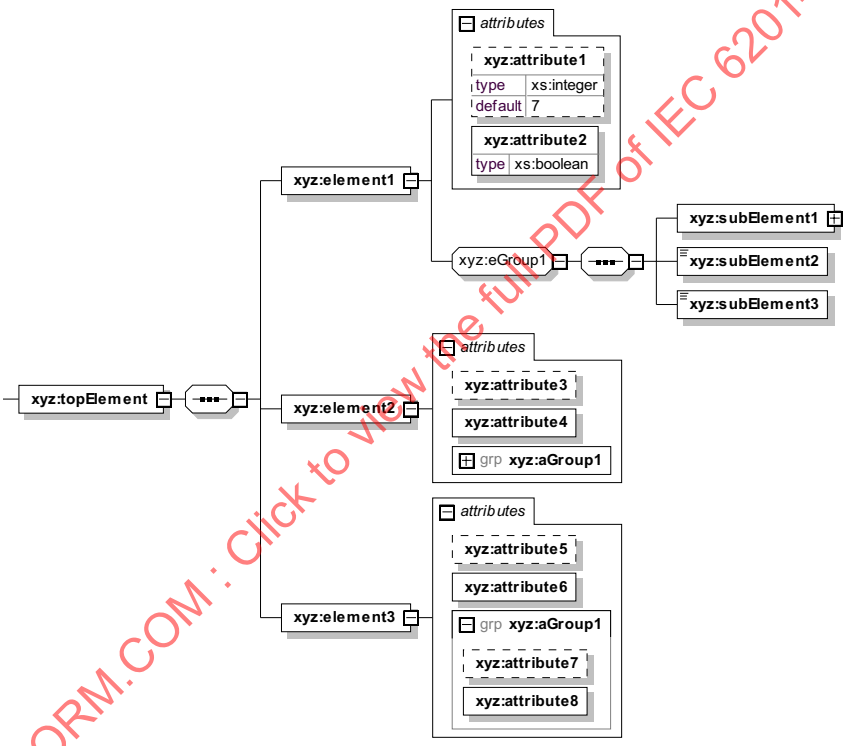


Figure 6—Attributes, groups, and attributeGroups

a) ***eGroup1*** is an element group inside **element1**. This group contains three subelements and the group symbol can be replaced by a solid line. The name of the group has no representation in the resulting output description. An element group can be optional, as indicated by a dashed outline (not shown) and it can also have a range, as indicated by numbers below the group symbol (not shown).

b) ***aGroup1*** is an ***attributeGroup*** inside **element2** and **element3**. This ***attributeGroup*** contains two attributes, **attribute7** and **attribute8**. Inside **element2**, the ***attributeGroup*** is shown in its collapsed form, as indicated by the small plus sign (+) inside the small box. Inside **element3** the ***attributeGroup*** is shown in it expanded form, as indicated by the small minus sign (−) inside the small box. **element2** contains four attributes: **attribute3**, **attribute4**, **attribute7**, and **attribute8**. **element3** also contains four attributes: **attribute5**, **attribute6**, **attribute7**, and **attribute8**. The name of the ***attributeGroup*** has no representation in the resulting description.

1.5.4.5.4 Wildcards

Figure 7 shows the use of wildcards. A *wildcard* is depicted by the rounded box with the **any ##any** text. Wildcards indicate that any well-formed attribute or element may be inserted into the containing element.

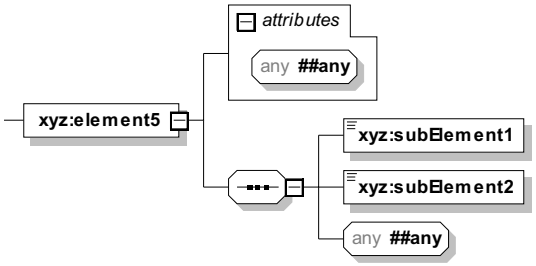


Figure 7—Wildcards

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, acronyms, and abbreviations used throughout the different specifications contained in this standard.

— Clause 4 defines the interoperability use model.

— Clause 5 defines the bus and abstraction definitions.

— Clause 6 defines the component and interconnect models.

— Clause 7 defines the designs and their connections.

— Clause 8 defines the abstractor model between abstraction definitions.

— Clause 9 defines the generator chain.

— Clause 10 defines the design and generator chain configuration.

— Clause 11 defines addressing and data visibility.

— Annexes. Following Clause 11 are a series of annexes.

2. Normative references

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

Simple Object Access Protocol (SOAP) version 1.2 specification, available from the W3C Web site at http://www.w3.org/TR/2007/REC-soap12-part1-20070427/.

Web Services Description Language (WSDL) version 1.1 specification, available from the W3C Web site at http://www.w3.org/TR/wsdl.

XML schema specification, available from the W3C Web site at 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.

XML version 1.0 specification, available from the W3C Web site at http://www.w3.org/TR/2000/REC-xml-20001006.

XPATH version 1.0 specification, available from the W3C Web site at http://www.w3.org/TR/1999/REC-xpath-19991116.

XSLT version 1.0 specification, available from the W3C Web site at http://www.w3.org/TR/1999/REC-xslt-19991116.

3. Definitions, acronyms, and abbreviations

For the purposes of this document, the following terms and definitions apply. *The IEEE Standards Dictionary: Glossary of Terms & Definitions* should be referenced for terms not defined in this clause.[12]

3.1 Definitions

abstraction definition: An object that describes a representation 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.

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

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

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

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

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

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, supports memory mapping and remapping, and can only have direct interfaces. *Syn*: **bus bridge**.

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

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

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.

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 remapping.

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**.

configurable element: An element in an IP-XACT description that can be set to a new value by a user, generator, or dependency equation. This includes all elements with a **resolve** attribute.

[12] *The IEEE Standards Dictionary: Glossary of Terms & Definitions* is available at http://shop.ieee.org/.

configurable IP: **IP** that contains **configurable element**s and an IP-specific generator capable of creating new components from the configured component and updating the design with the new version of the **component**. *Syn*: **configurable component**.

configuration manager: An object that creates and manages top-level meta-description of **system on 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.

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

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 port becomes stable in a certain time period and remains stable for a certain time period relative to a particular clock edge.

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

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.

design configuration: This description 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 description and can specify a **view** for the **component** instances and **abstractor**s for each **interconnection**, and configure generator **chain**s. *Syn*: **configuration**.

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

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 reuse-centric development flow is efficiently enabled. This is managed by creating and maintaining a meta-data description of the **SoC**.

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.

electronic design intellectual property (**IP**): Used in the electronic design community to refer to a reusable collection of design specifications that represent the behavior, properties, and/or representation of the design in various media. The name IP is partially derived from the common practice of considering a collection of this type to be the intellectual property of one party. Both hardware and software collections are encompassed by this term. IP utilized in the context of a SoC design or design flow may include specifications; design models; design implementation descriptions; 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 and configurable IP.

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

eXtensible Markup Language (**XML**): A simple, very flexible text format derived from SGML.

NOTE—See ISO/IEC 8879 [B12].[13]

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

fixed IP: **IP** that has no elements that are configured by the **DE** or set by industry de facto tools.

generator: Combines **component meta-data** with **architectural rules** to provide a consistent system description that 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.

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**.

generator chain: A hierarchical list of generators used to define the order for executing **generator**s. A design flow can be represented by a generator chain.

generator group: A symbolic name assigned to a **generator** to enable generator selection.

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

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 that relate to the meta-data description.

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 description of which contains 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.

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

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

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`.

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

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.

[13]Notes in text, tables, and figures are given for information only and do not contain requirements needed to implement the standard.

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

initiative: An abstract description of port modes: requires, provides, both, or phantom. Used for transactional level modeling (TLM).

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

interface connection: Bus 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**.

IP generators: Tools that 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.

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

IP platform architect: Creator of platform-based architectures.

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

IP repository: Database of IP.

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

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

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

memory map: A block of memory in a **component** (which may be accessible through a **slave** interface).

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

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

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.

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

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

objects: XML descriptions of the following types: **component**s, **design**s, **busDefinition**s, **abstractionDefinition**s, **abstractor**s, **designConfiguration**s, and **generatorChain**s. To be able to be uniquely referenced, each object has an unique identifier called its **Vendor Library Name Version** (**VLNV**).

opaque bridge: A bus interconnect component that may modify the address space of a master bus interface of one bus type to the memory map of a slave bus interface of another bus type and does not allow direct access to any components residing on that address space. An opaque bridge has the **opaque** attribute equal to **true**.

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.

NOTE—See Transaction-Level Model of SystemC [B13].

phantom port: An initiative of a port that indicates this port does not have a true connection to the implementation, e.g., the port does not appear on the VHDL entity.

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

platform: Architectural (sub)system framework.

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

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

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

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.

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

segment: A portion of an **addressSpace**, defined with an address offset and range.

semantic consistency rules (**SCR**s): Additional rules applied to an XML description that cannot be expressed in the **schema**. Typically, these are rules between elements in multiple XML descriptions.

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.

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

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

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

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

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).

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

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

tool plug-ins: Tools that 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**.

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.

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

transparent bridge: A bus interconnect component that modifies the address space of a master bus interface of one bus type to the memory map of a slave bus interface of another bus type with directly addressable access to any components residing on that address space. A transparent bridge has the **opaque** attribute equal to **false**.

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

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

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

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

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

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

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

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

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

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

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

XSLT: XSL Transform is a particular program written in the XSL language for performing a transformation (from one version to the next).

3.2 Acronyms and abbreviations

AHB	AMBA® high-speed bus[14]
APB	AMBA peripheral bus
API	application programmers interface
AXI	Advanced eXtensible Interface
DE	design environment
EDA	electronic design automation
ESL	electronic system level
HDL	hardware description language
IP	electronic design intellectual property
LAU	least addressable unit (of memory)
OSCI	Open SystemC Initiative
PV	programmer's view
PVT	programmer's view with timing
RAM	random access memory
ROM	read only memory
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
XSLT	XSL Transform
XML	eXtensible Markup Language
3MD	three levels of meta-data

[14]AMBA is an open specification on-chip backbone for interconnecting **intellectual property** (IP) blocks. AMBA is a registered trademark of ARM Limited. This information is given for the convenience of users of this standard and does not constitute an endorsement by the IEEE of this product. Equivalent products may be used if they can be shown to lead to the same results.

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. All or some of the roles can be mixed within a single organization, e.g., some electronic design automation (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 standard, the roles and responsibilities are restricted to the scope of IP-XACT v1.5 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 namespace 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 the following 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.

— *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, done with VHDL generics or Verilog parameters.

— *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 that has a 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 subsystems. 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 subsystems. 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, floor-planing, 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 support specialized design-automation features, such as architectural-rule checking.

4.2 IP-XACT IP exchange flows

This subclause 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, as follows:

— 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 description tree (which is composed of files of different types: HDL source files, data sheets, interfaces, parameters, etc.) and package 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

The system design tool takes IP-XACT components and designs as input, configures them, and loads them into its own database format. Then it can automate some tasks, such as creating the platform, generating the component interconnect and 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 IP-XACT documents. In this use model, each SoC design tool uses its own generators (possibly utilizing the IP-XACT TGI) to build and update its internal meta-data state and export to an IP-XACT format. Then the IP-XACT description can be imported by another IP-XACT enabled EDA tool.

5. Interface definition descriptions

5.1 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 descriptions, a bus definition and an abstraction definition. These two descriptions are referenced by components or abstractors in their bus or abstractor interfaces.

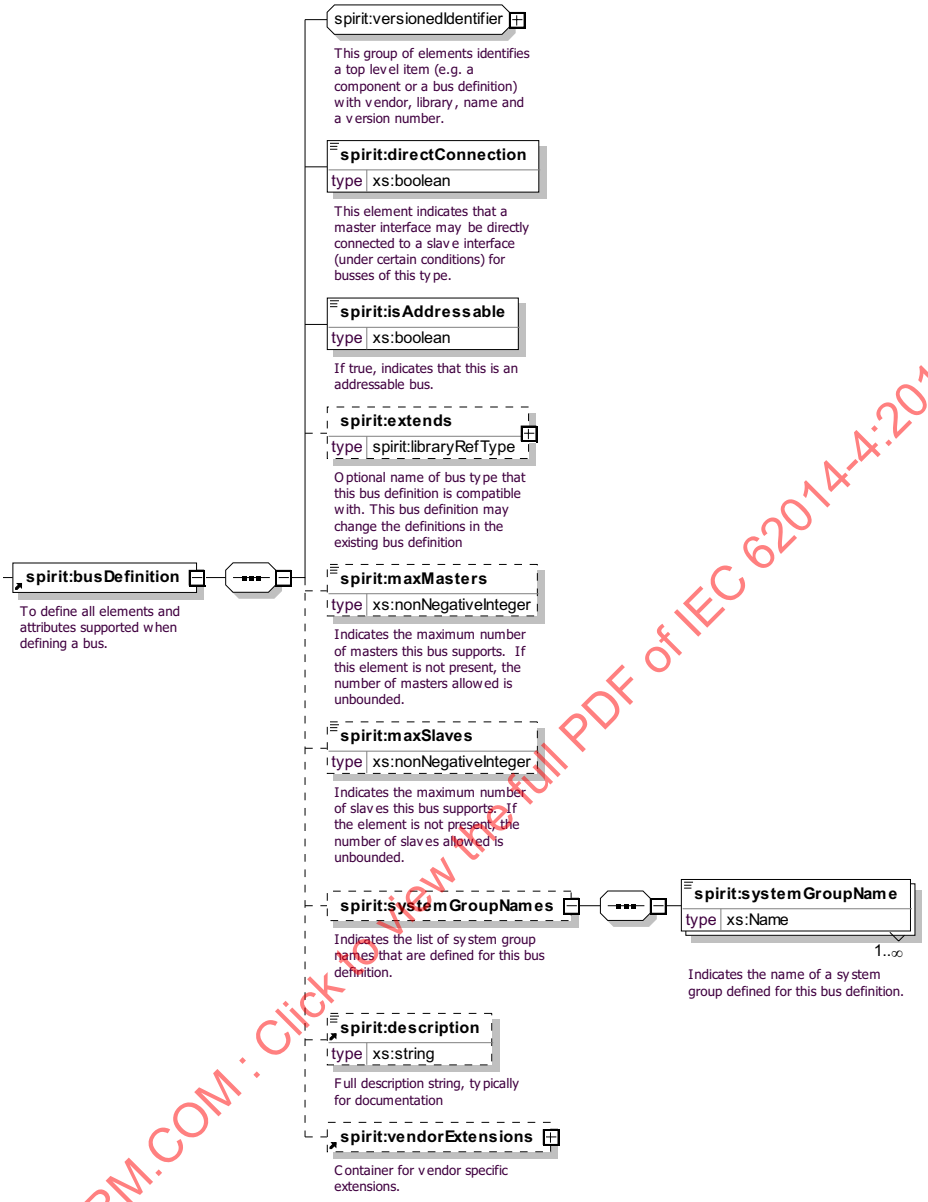
The *bus definition* description contains the high-level attributes of the interface, including items such as the connection method and indication of addressing.

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. See 6.5.

5.2 Bus definition

5.2.1 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.



5.2.2 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. See C.6

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 (i.e., master–mirroredMaster, slave–mirroredSlave, or system–mirroredSystem).

c) **isAddressable** (mandatory) specifies the bus has addressing information. The **isAddressable** element is of type ***boolean*** (see 6.3). 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 C.7), it contains four attributes to specify a unique VLNV. See also: 5.12.

e) **maxMasters** specifies the maximum number of masters that are allowed on the 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 are allowed to appear on the 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 **systemGroupName** elements, which in turn 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 **systemGroupName** shall be unique with the containing **systemGroupNames** element. The **systemGroupName** 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 C.10.

See also: SCR 1.3, SCR 1.9, SCR 1.11, and SCR 6.17.

5.2.3 Example

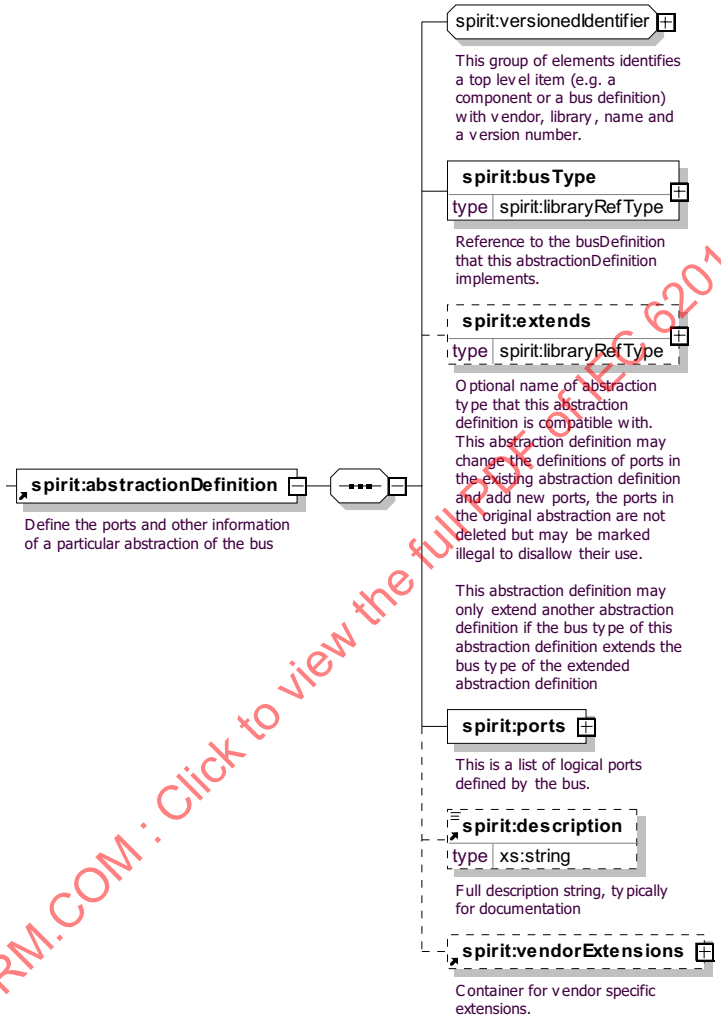
This is an example of an AHB **busDefinition**.

```
<?xml version="1.0" encoding="UTF-8" ?>
<spirit:busDefinition
xmlns:spirit= http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5
http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5/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>
```

5.3 Abstraction definition

5.3.1 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.



5.3.2 Description

The **abstractionDefinition** element describes 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. See C.6.

b) **busType** (mandatory) specifies the bus definition that this abstraction defines. The **busType** element is of type ***libraryRefType*** (see C.7); it contains four attributes to specify a unique VLNV. See also: 5.12.

c) **extends** (optional) specifies if this definition is an extension from another abstraction definition. The **extends** element is of type ***libraryRefType*** (see C.7), it contains four attributes to specify a unique

VLNV. 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: 5.12.

d) **ports** (mandatory) is a list of logical ports, see 5.4.

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. See C.10.

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 5.7) or a transactional port (see 5.10). 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.

See also: SCR 1.9, SCR 1.11, SCR 1.13, SCR 3.1, SCR 3.16, SCR 3.17, and SCR 6.11.

5.3.3 Example

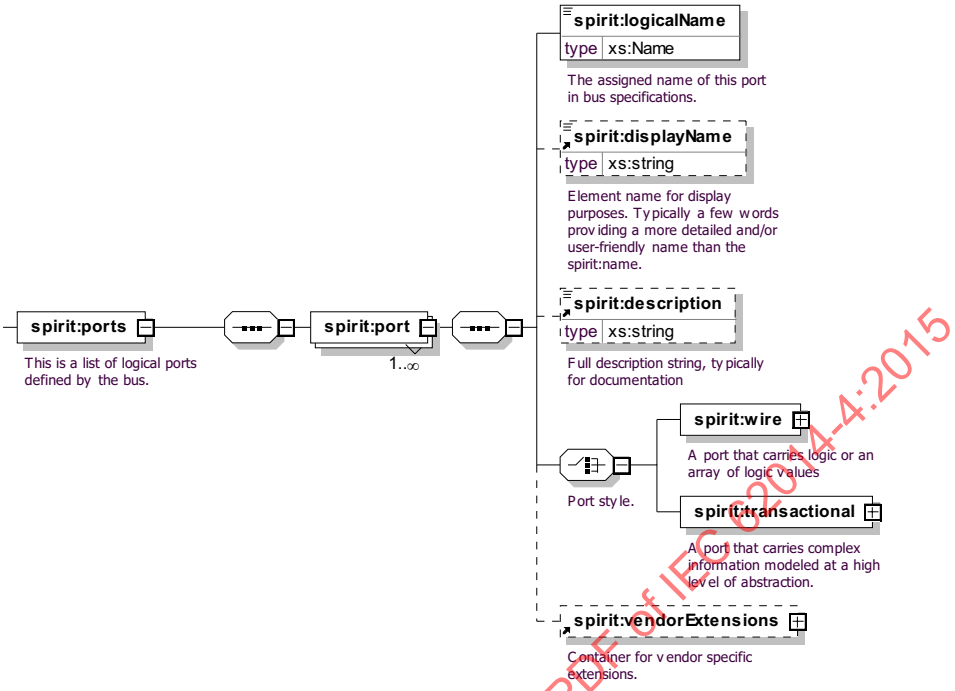
The following example shows an abstraction definition for the interrupt bus in the Leon2 TLM example.

```
<spirit:vendor>spiritconsortium.org</spirit:vendor>
<spirit:library>Leon</spirit:library>
<spirit:name>INT_PV</spirit:name>
<spirit:version>1.5</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>
```

5.4 Ports

5.4.1 Schema

The following schema details the information contained in the **ports** element, which appears as part of the **abstractionDefinition** element within an abstraction definition. This is different from the **ports** element that appears as part of the **model** element within components.



5.4.2 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 **logicalName** shall be unique within the **abstractionDefinition**. 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 5.5 or 5.10, 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. See C.10.

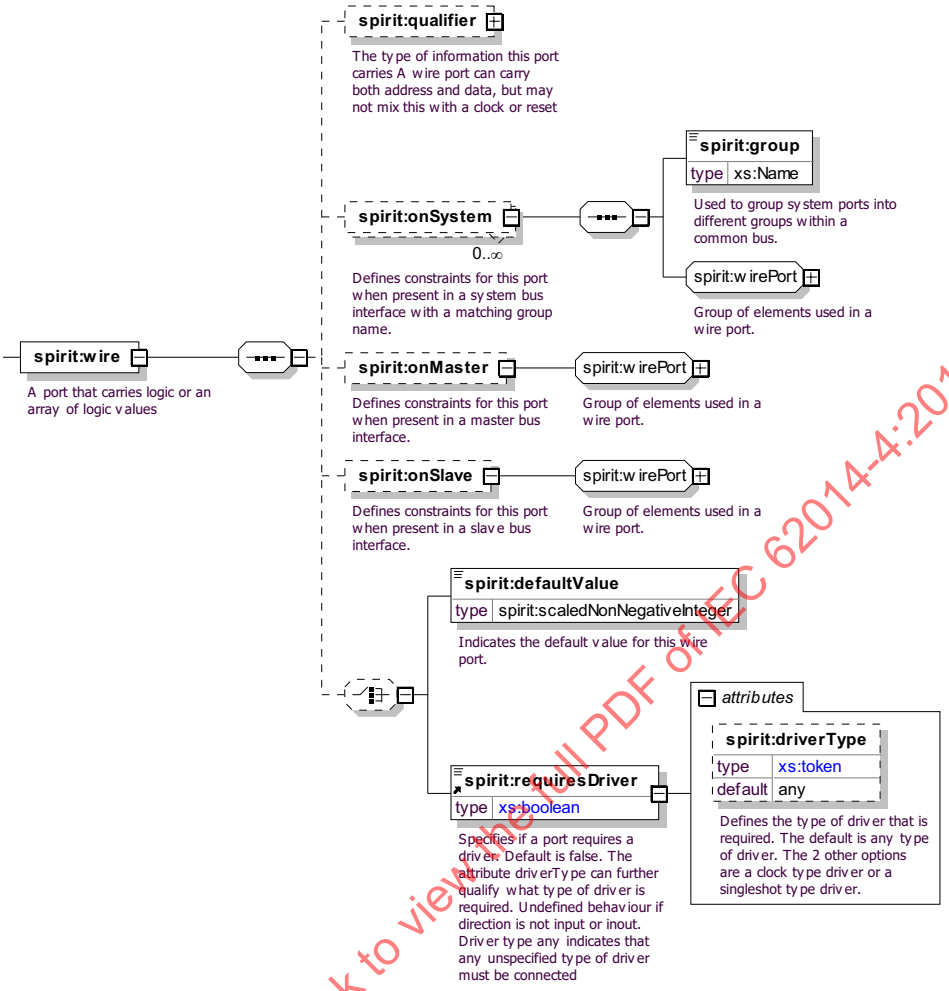
5.4.3 Example

See 5.3.3 for an example.

5.5 Wire ports

5.5.1 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**).



5.5.2 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 5.6.

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 type of this element is ***Name***.

   2) The group ***wirePort*** specifies what elements are used in this port. See 5.7.

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 5.7.

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 5.7.

e) Either of the following two elements are allowed, but not both.

1) **defaultValue** (optional) contains the default logic value for this wire port. This value is applied when the **busInterface** is connected, this logical port is not connected, and there is not an ad hoc connection to the corresponding physical port. 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***. If this element is not present, its effective value is **false**, indicating this port does not require a driver. When set to **true**, the attribute **driverType** further qualifies what driver type is required: **any** (meaning any logic signal or value), **clock** (meaning a repeating type waveform), or **singleshot** (a non-repeating type waveform). If this element is not present, its effective value is **any**.

NOTE—The **onMaster**, **onSlave**, and **onSystem** elements associated with each logical port provide optional constraints. If any of these are missing, there are no constraints for how the port appears on interfaces with that mode (master, system, or slave). A port may appear in any system interface group unless its presence is marked as illegal for that group. 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 IP with interface that conforms to the abstraction definition.

See also: SCR 6.12 and SCR 6.15.

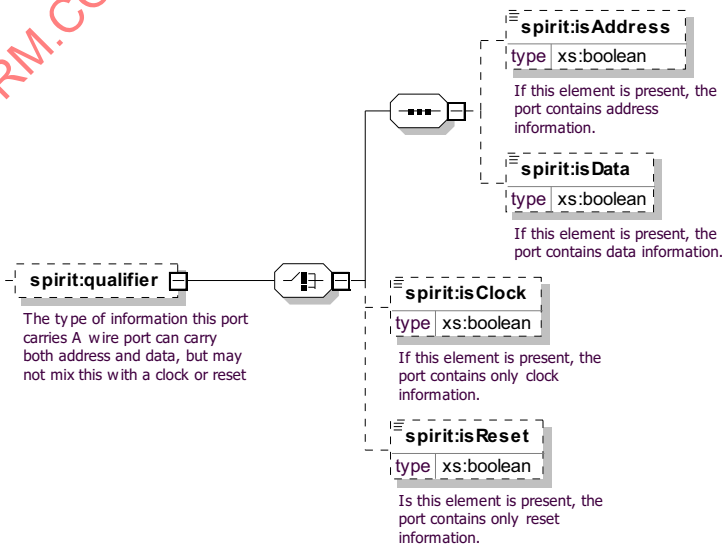
5.5.3 Example

See 5.3.3 for an example.

5.6 Qualifiers

5.6.1 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**).



5.6.2 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 (e.g., used with serial protocols). The type of this element is ***boolean***. See also: Clause 11.

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 (e.g., used with serial protocols). The type of this element is ***boolean***. See also: Clause 11.

c) **isClock** (optional), when **true**, specifies this port is a clock for this bus interface, i.e., it provides a repeating pattern that 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 ports. This **qualifier** shall not be combined with other qualifiers. The type of this element is ***boolean***.

d) **isReset** (optional), when **true**, specifies this port 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 ports. This **qualifier** shall not be combined with other qualifiers. The type of this element is ***boolean***.

See also: SCR 6.17, SCR 9.1, SCR 9.2, and SCR 12.8.

5.6.3 Example

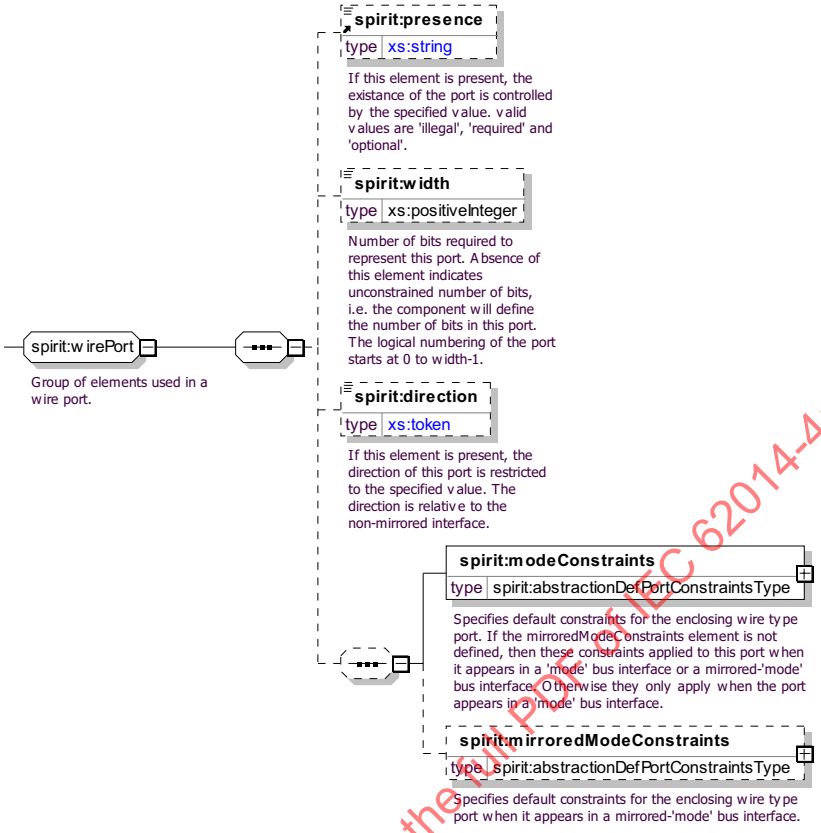
```
<spirit:port>
   <spirit:logicalName>Clock</spirit:logicalName>
   <spirit:wire>
      <spirit:qualifier>
         <spirit:isClock>true</spirit:isClock>
      </spirit:qualifier>
      <spirit:onSystem>
         <spirit:group>clk</spirit:group>
         <spirit:width>1</spirit:width>
         <spirit:direction>out</spirit:direction>
      </spirit:onSystem>
      <spirit:onMaster>
         <spirit:direction>in</spirit:direction>
      </spirit:onMaster>
      <spirit:onSlave>
         <spirit:direction>in</spirit:direction>
      </spirit:onSlave>
   </spirit:wire>
</spirit:port>
<spirit:port>
   <spirit:logicalName>Resetn</spirit:logicalName>
   <spirit:wire>
      <spirit:qualifier>
         <spirit:isReset>true</spirit:isReset>
      </spirit:qualifier>
      <spirit:onSystem>
         <spirit:group>reset</spirit:group>
         <spirit:width>1</spirit:width>
         <spirit:direction>out</spirit:direction>
```

```
            </spirit:onSystem>
            <spirit:onMaster>
               <spirit:direction>in</spirit:direction>
            </spirit:onMaster>
            <spirit:onSlave>
               <spirit:direction>in</spirit:direction>
            </spirit:onSlave>
         </spirit:wire>
      </spirit:port>
      <spirit:port>
         <spirit:logicalName>Address</spirit:logicalName>
         <spirit:wire>
            <spirit:qualifier>
               <spirit:isAddress>true</spirit:isAddress>
            </spirit:qualifier>
            <spirit:onMaster>
               <spirit:direction>out</spirit:direction>
            </spirit:onMaster>
            <spirit:onSlave>
               <spirit:direction>in</spirit:direction>
            </spirit:onSlave>
         </spirit:wire>
      </spirit:port>
```

5.7 Wire port group

5.7.1 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***).



5.7.2 Description

The ***wirePort*** group 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**. If this element is not present, its effective value is **optional**.

b) **width** (optional) represents the number of logical bits that are required to represent this port. 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 port, 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 (see 6.2). The three possible values are **in**, **out**, 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 5.8). Any **mirroredModeConstraints** apply to this port if it appears in a mirrored-*mode* bus interface (see 5.9).

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

See also: SCR 6.5, SCR 6.6, SCR 6.7, and SCR 6.18.

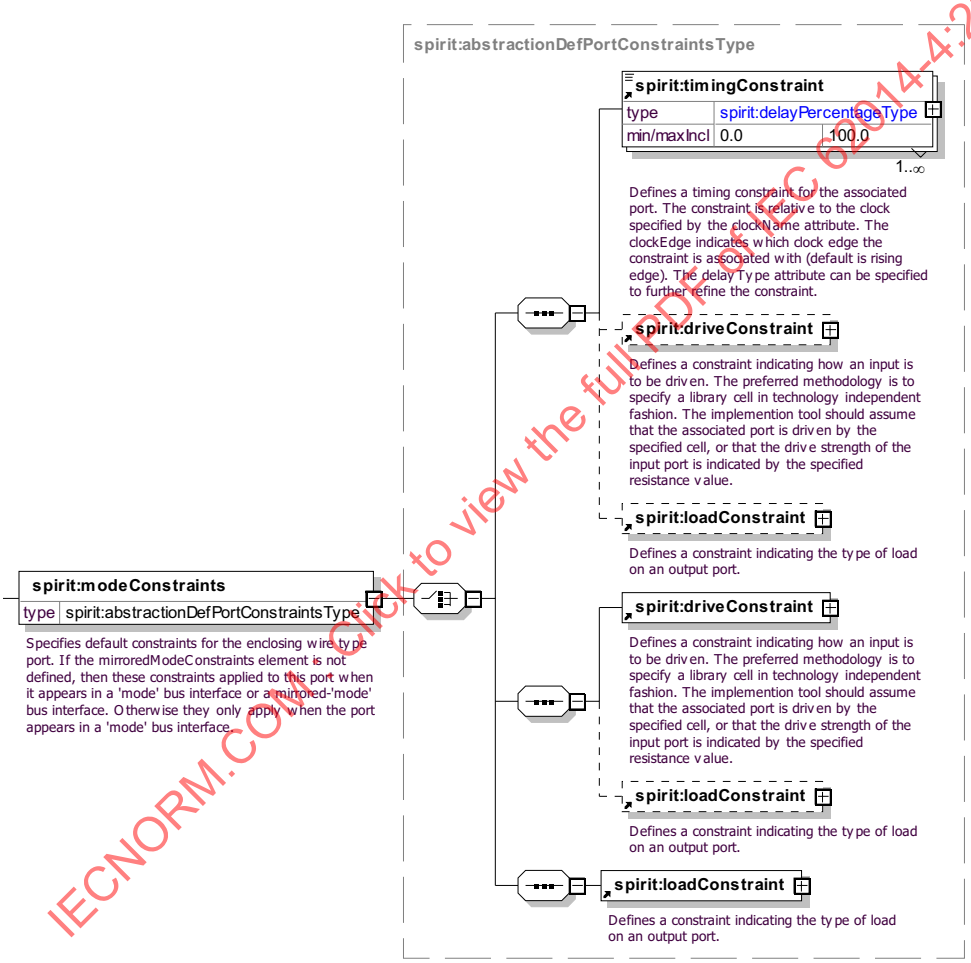
5.7.3 Example

See 5.3.3 for an example.

5.8 Wire port *mode* constraints

5.8.1 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***).



5.8.2 Description

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

a) **timingConstraint** (optional) element defines a technology-independent timing constraint associated with the containing wire port. See 6.11.13.

b) **driveConstraint** (optional) element defines a technology-independent drive constraint associated with the containing wire port. See 6.11.12.

c) **loadConstraint** (optional) element defines a technology-independent load constraint associated with the containing wire port. See 6.11.11.

The constraints contained within the **modeConstraints** element are only applied to the corresponding physical ports in a component when the physical port does not have any constraints defined within its own port element and there is no standard design constraint (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 5.9), 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.

5.8.3 Example

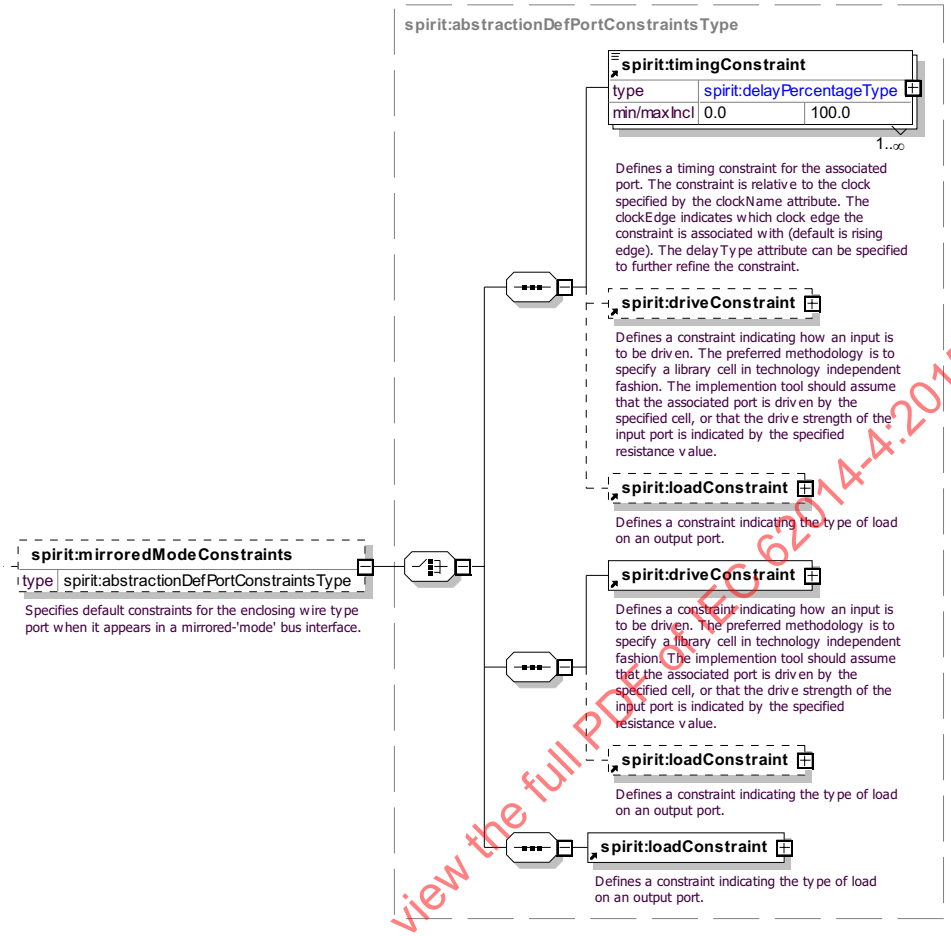
The following example shows a port within an abstraction definition, containing a single timing constraint. Since there is no **mirroredModeConstraint** element, this timing constraint applies when the HRDATA port appears in either a master interface or a mirrored-master interface. On a master interface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

5.9 Wire port mirrored-*mode* constraints

5.9.1 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***).



5.9.2 Description

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

a) **timingConstraint** (optional) element defines a technology-independent timing constraint associated with the containing wire port. See 6.11.13.

b) **driveConstraint** (optional) element defines a technology-independent drive constraint associated with the containing wire port. See 6.11.12.

c) **loadConstraint** (optional) element defines a technology-independent load constraint associated with the containing wire port. See 6.11.11.

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.

5.9.3 Example

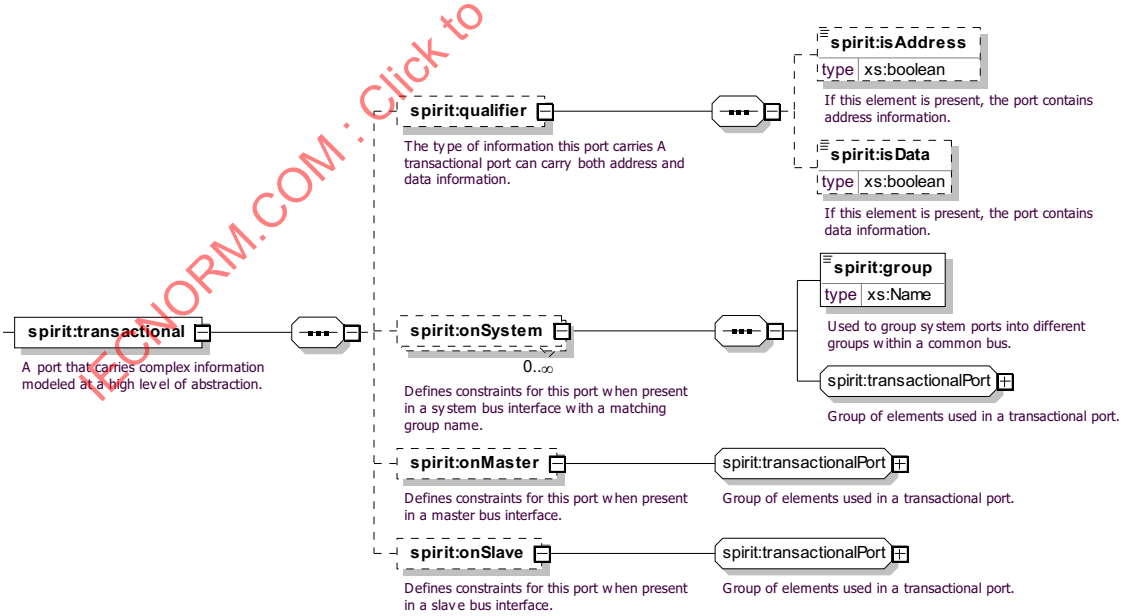
The following example shows a port within an abstraction definition, containing a single timing constraint. On a master interface the port gets 40% of the cycle time and on a mirrored master interface it gets 50% of the cycle time.

```
<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”>50
               </spirit:timingConstraint>
         </spirit:mirroredModeConstraints>
      </spirit:onMaster>
   /spirit:wire>
</spirit:port>
```

5.10 Transactional ports

5.10.1 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**).



5.10.2 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** (optional) 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 **systemGroupName**.

   2) The group ***transactionalPort*** specifies what elements are used in this port. See 5.11.

c) **onMaster** (optional) 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 5.11.

d) **onSlave** (optional) 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 5.11.

NOTE—The **onMaster**, **onSlave**, and **onSystem** elements associated with each logical port provide optional constraints. If any of these are missing, there are no constraints for how the port appears on interfaces with that mode (master, system, or slave). If no **onSystem** constraint is specified with a particular group, there are no constraints for system interfaces in that group. 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 IP with interface that conform to the abstraction definition.

See also: SCR 6.13, SCR 6.14, SCR 6.15, and SCR 6.17.

5.10.3 Example

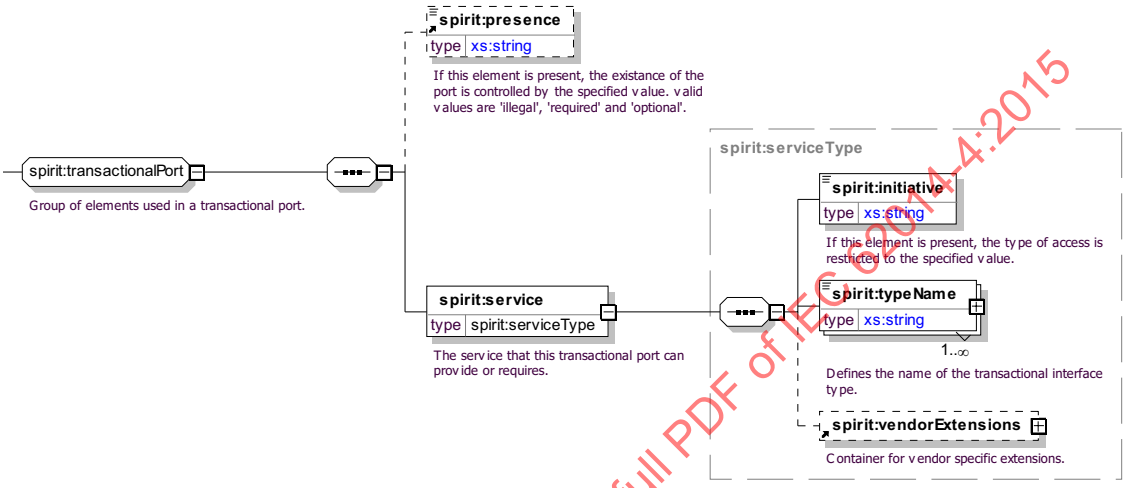
The following example shows a transactional port within an abstraction definition, carrying data information.

```
<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>
```

5.11 Transactional port group

5.11.1 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***).



5.11.2 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**. If this element is not present, its effective value is **optional**.

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**, **provides**, or **both**. For example, a SystemC `sc_port` is defined using **requires**, since it requires a SystemC interface.

   2) **typeName** (mandatory) is an 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 C.10.

See also: SCR 6.2, SCR 6.3, SCR 6.4, SCR 6.8, and SCR 6.18.

5.11.3 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>
```

5.12 Extending bus and abstraction definitions

5.12.1 Extending bus definitions

Bus definitions may use the **extends** element to create a family of compatible interconnectable 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 interconnectable, they need to be in a direct line of descent in the hierarchical *extension* tree, as illustrated in Figure 8.

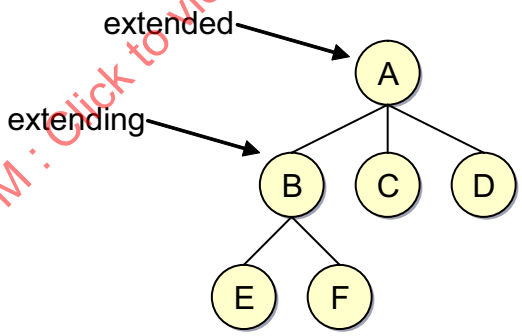


Figure 8—Extends relation hierarchy tree

In Figure 8, 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`.

5.12.2 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 9.

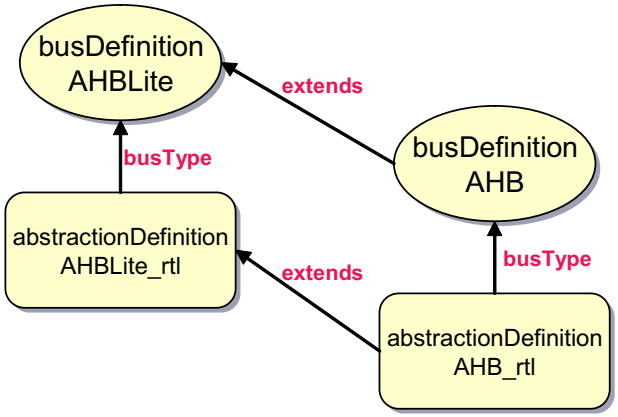


Figure 9—Example of extending

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.

See also: SCR 3.17.

5.12.3 Modifying definitions

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

Table 1—Elements of *extending* bus definition

Item	Modified	Comment
directConnection	No	
isAddressable	No	
maxMasters	Yes	Smaller number applies when connecting interfaces of extended bus definitions.
maxSlaves	Yes	Smaller number applies when connecting interfaces of extended bus definitions.
systemGroupNames	Yes	New group names may be added; group names not specified are not allowed by this bus definition.
description	Yes	
vendorExtensions	Yes	

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

Table 2—Elements of *extending* abstraction definition

Item	Modified	Comment
ports	Yes	See Table 3 and SCR 6.11.
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 from 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	This default can be used to set a value for the extended abstraction definition logical port, if this port is not mapped or its presence is marked as illegal.
service/initiative	No	
service/typeName	No	
service/vendorExtensions	Yes	
vendorExtensions	Yes	

5.12.4 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 modifications of the port list (i.e., adding or removing ports). The abstraction definition writer needs to be aware of these scenarios and specify default values where necessary. Following are 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`)

system(`A`) connecting to mirrored-system(`B`)

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

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

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

system(`B`) connecting to mirrored-system(`A`)

5.13 Clock and reset handling

Abstraction definitions shall include all the logical ports that can participate in the protocol of the bus; bus interfaces also 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 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 reset ports 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, does not need to exist as a port of the component implementation, since it may be mapped to a phantom port of the component (see 6.11.18.2). 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 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.

6. Component descriptions

6.1 Component

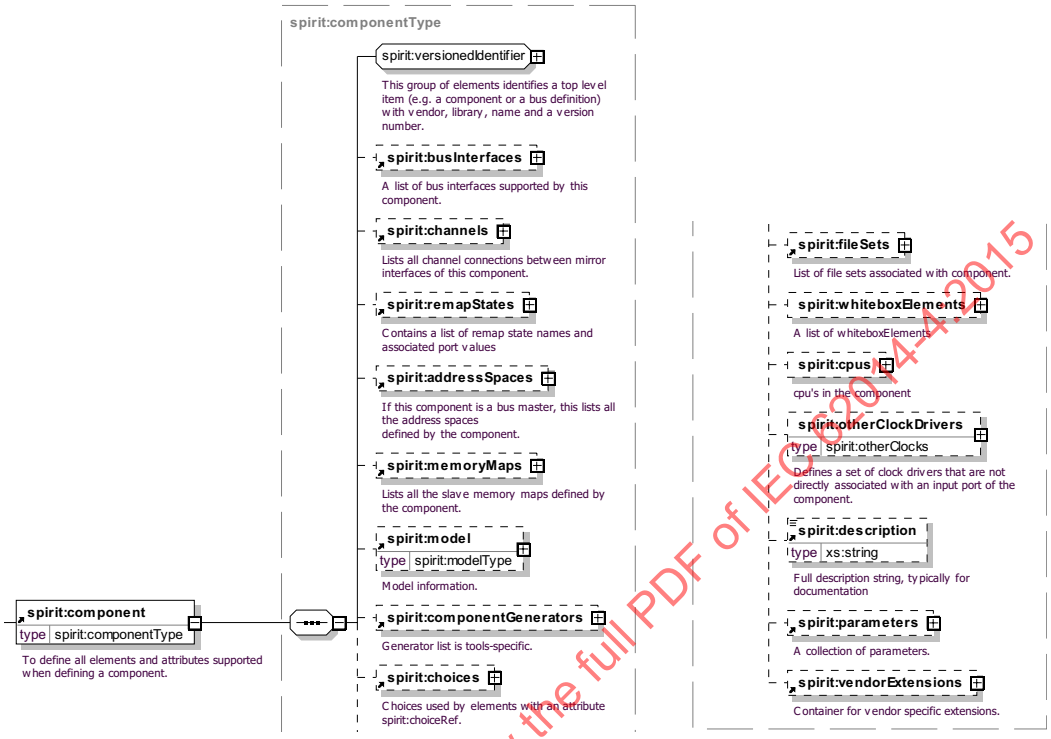
An IP-XACT *component* is the central placeholder for the objects meta-data. Components are used to describe cores (processors, co-processors, DSPs, etc.), peripherals (memories, DMA controllers, timers, UART, etc.), and buses (simple buses, multi-layer buses, cross bars, network on chip, etc.). An IP-XACT component can be of two kinds: static or configurable. A DE cannot change a *static component*. A *configurable* (or *parameterized*) *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.

6.1.1 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.



6.1.2 Description

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

a) ***versionedIdentifier*** group provides a unique identifier; it consists of four subelements for a top-level IP-XACT element. See C.6.

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 6.5.

c) **channels** (optional) specifies the interconnection between interfaces inside of the component. See 6.6.

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 6.9.2.

e) **addressSpaces** (optional) specifies the addressable area as seen from **busInterface**s with an interface mode of **master** or from **cpu**s. See 6.7.

f) **memoryMaps** (optional) specifies the addressable area as seen from **busInterface**s with an interface mode of **slave**. See 6.8.

g) **model** (optional) specifies all the different views, ports, and model configuration parameters of the component. See 6.11.

h) **componentGenerators** (optional) specifies a list of generator programs attached to this component. See 6.12.

i) **choices** (optional) specifies multiple enumerated lists. These lists are referenced by other sections of this component description. See 6.14.

j) **fileSets** (optional) specifies groups of files and possibly their function for reference by other sections of this component description. See 6.13.

k) **whiteboxElements** (optional) specifies all the different locations in the component that can be accessed for verification purposes. See 6.15.

l) **cpus** (optional) indicates this component contains programmable processors. See 6.17.

m) **otherClockDrivers** (optional) specifies any clock signals that are referenced by implementation constraints, but are not external ports of the component. See 6.11.15.

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 C.11.

p) **vendorExtensions** (optional) contains any extra vendor-specific data related to the component. See C.10.

See also: SCR 1.9.

6.1.3 Example

This is an example of a component (a `timers` peripheral in a `Leon2` library).

```
<?xml version="1.0" encoding="UTF-8" ?>
<spirit:component
    xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5
    http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5/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>
```

6.2 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 5.2)] and an abstraction type [i.e., a reference to an **abstractionDefinition** (see 5.3)]. The purpose of the bus interface is to map the physical ports 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: a bus interface may be a master, slave, or system interface, and may be direct or mirrored. The seventh interface mode is the monitor mode. A monitor interface can be used to connect IP into the design for verification.

6.2.1 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 programmer's view (PV)].

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 participates in the protocol of the master or slave interfaces, it shall be included in master and slave interfaces. System interfaces often contain some of the same ports as master or slave 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 stand-alone), these ports should have some sort of **system** element designator in the bus definition.

6.2.2 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 direct counterpart) supports the master, slave, and system interface modes.

6.2.3 Monitor interface modes

A *monitor interface* connects 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.).

6.3 Interface interconnections

IP-XACT provides 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 7.1.

6.3.1 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 component containing a **bridge** element(s).

See also: SCR 2.2, SCR 2.10, SCR 2.11, SCR 2.12, SCR 2.13, and SCR 2.14.

6.3.2 Mirrored-non-mirrored connection

A mirrored-non-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 mirrored-non-mirrored connections are possible with the use of a component containing a **channel** element.

See also: SCR 2.2, SCR 2.12, and SCR 2.14.

6.3.3 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.

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

b) The direction or initiative of ports in a monitor interface cannot be specified in an abstraction definition. All wire ports on a monitor interface shall be treated as having a logical direction of **in**. A monitor interface connected to any active interface shall see the values on the wire ports of the active interface as inputs on its ports regardless of the direction they have on the active interface. All transactional ports on a monitor interface shall be treated as having a logical initiative of **requires**.

See also: SCR 2.2, SCR 4.6, and the SCRs in Table B.4.

6.3.4 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 a 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 `width-1` down to `0` if the width is present. If the width is not present, the logical port number shall have a lower bound of `0` and does not have an upper bound.

6.3.4.1 Mapping rules

Mapping rules describe the assignment of logical bit numbers to physical bit numbers.

a) First, apply all the rules defined in B.1.8 to determine the logical and physical ranges.

b) The mapping is logical.left-> physical.left down to logical.right-> physical.right.

6.3.4.2 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.

6.4 Complex interface interconnections

There are two constructs used to connect interfaces of standard components together (traditional components, usually with *master* and *slave* interfaces), a channel and a bridge. These constructs are encapsulated into components. Not only does the **channel** or **bridge** component provide a connection between 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* identifies interfaces in a component that connect a component's master, slave, and system interfaces on the same bus. All masters connected to a channel see each slave at the same physical address. On a channel, only one master may initiate transactions at a time. This does not preclude bus protocols that utilize pipelining or out-of-order completion. A bus that has addresses that are simultaneously seen differently from different masters or a bus that allows transactions from different masters to be simultaneously initiated may only be represented using bus bridges, not channels.

A *bridge* is an interface between two separate buses, which may be of the same or different types. 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 component containing bus bridges—such examples might have multiple master and multiple slave interfaces.

6.4.1 Channel

A *channel* is a general name that 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 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).

b) A channel supports memory mapping and remapping (see 6.8, 6.9, Clause 10, and H.3).

See also: SCR 3.2.

6.4.2 Bridge

Some buses can be modeled using a component as a bridge. A *bridge* is a component that physically links one or more master bus interfaces to a slave bus interface and logically connects the master address space(s) to a slave memory map having two bus types on each side. This component has at least one master bus interface and at least one slave bus interface, each for different protocols, and the bridge translates any signals between them. The slave bus's interface definition contains a **bridge** element (or a set of them) to designate the corresponding master bus interface(s). There are two different types of bridges defined in IP-XACT: transparent and opaque. See also: Annex H.

The bridge relationship is *transparent* (**opaque** attribute is **false**) when the address space on the bridged master bus interface is a decoded subset of the main address space, as seen through the bus bridge's slave bus interface. In this case, a slave component connected on the bridged master side shall reserve an address block on the main memory map seen on the bridging slave side. If nothing is attached to the bridged master bus interface, then no address block is reserved on the main memory map.

The bridge relationship is *opaque* (**opaque** attribute is **true**) when the address space on the bridged master bus interface is not directly accessible to the main address space as seen from the channel to which the slave bus interface is connected. In this case, the bridging component occupies a single address block, which is the size of its slave bus interface, reserved on the memory map of the masters attached to the main bus channel.

The following rules apply for using bridges.

a) A slave interface can bridge to multiple address spaces. Specifically, a bridge shall have one or more master interfaces and each master interface may have an 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 (e.g., master interface to slave interface connection) under the conditions defined in 6.3.1. Or it can connect to a channel (e.g., master interface to mirrored-master interface).

c) A bridge supports memory mapping and remapping (see 6.8, 6.9, Clause 10, and H.3).

The transfer of addressing information from the slave interface to the master interface of a bridge is done through the address space assigned to the master interface. This address space defines the visible address range from this master interface.

6.4.3 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 that supports 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 crossbar 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, as shown in Figure 10.

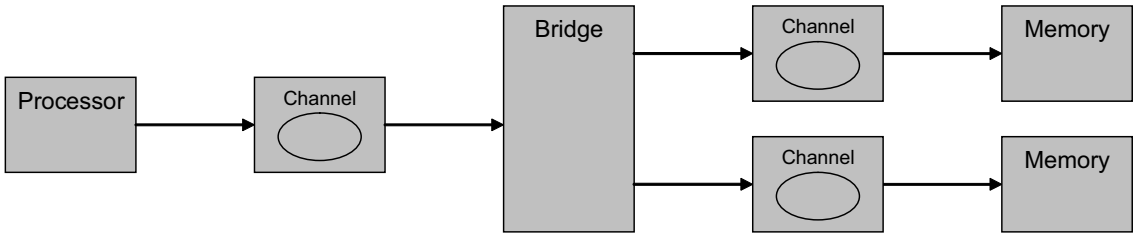


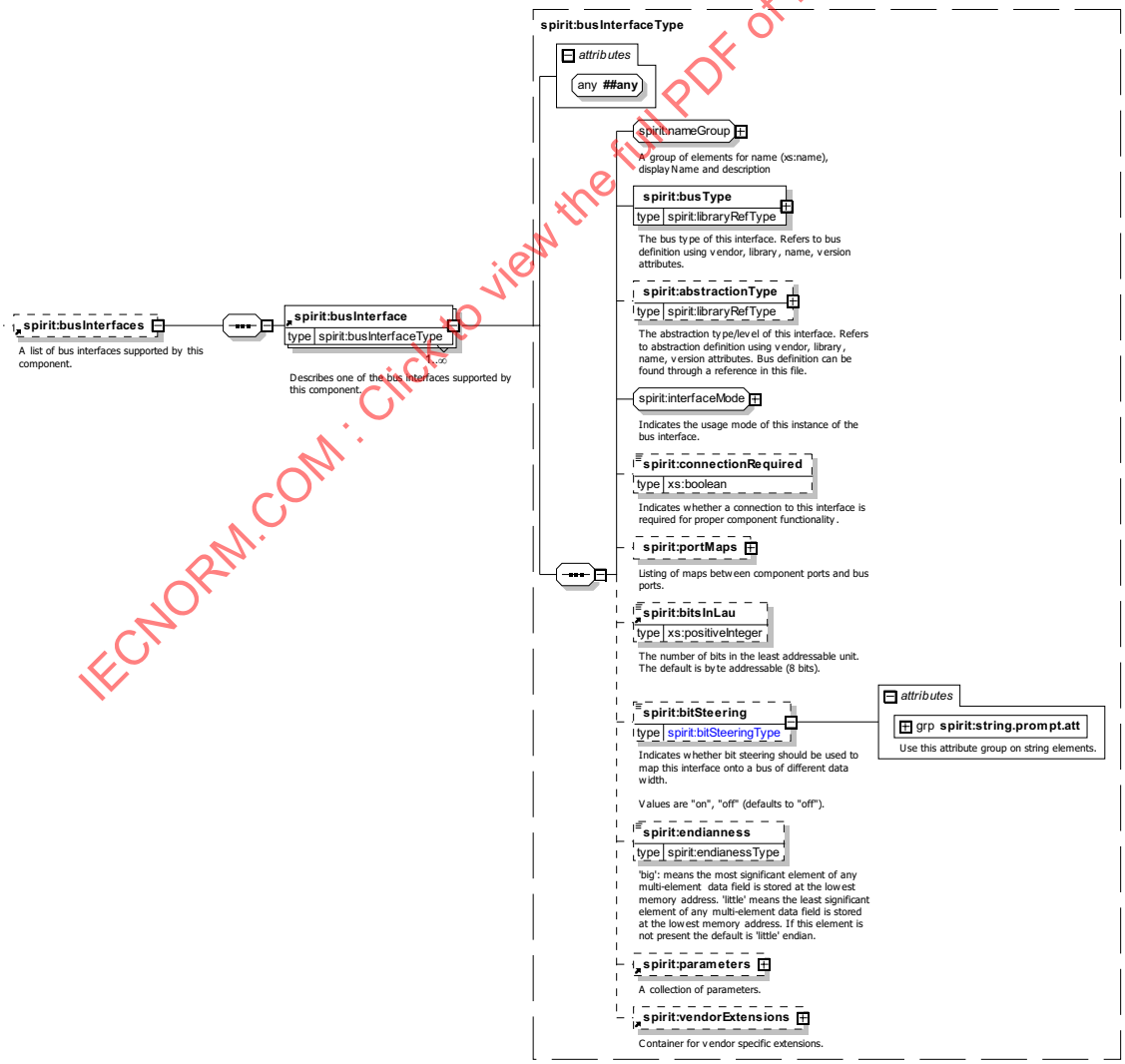
Figure 10—Asymmetric multi-layer bus connection using channels

6.5 Bus interfaces

6.5.1 busInterface

6.5.1.1 Schema

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



6.5.1.2 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. The **busInterface** element also allows for vendor attributes to be applied. It contains the following elements and attributes.

a) ***nameGroup*** group is defined in C.1. The **name** element shall be unique within the containing **component** element.

b) **busType** (mandatory) specifies the bus definition that this bus interface is referenced. A bus definition (see 5.2) describes the high-level attributes of a bus description. The **busType** element is of type ***libraryRefType*** (see C.7); it contains four attributes to specify a unique VLNV.

c) **abstractionType** (optional) specifies the abstraction definition where this bus interface is referenced. An abstraction definition describes the low-level attributes of a bus description (see 5.3). The **abstractionType** element is of type ***libraryRefType*** (see C.7); it contains four attributes to specify a unique VLNV.

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, mirrored-System and monitor. See 6.5.2.

e) **connectionRequired** (optional), if **true**, specifies when this component is integrated; this interface shall be connected to another interface for the integration to be valid. If **false**, this interface may be left unconnected. If this element is not present, its effective value is **false**. 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 6.5.6.

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 C.12.

i) **endianness** (optional) indicates the endianness of the bus interface. The two choices are **big** for big-endian and **little** for little-endian. If this element is not present, its effective value is **little**. See also 6.5.1.2.1.

j) **parameters** (optional) specifies any parameter data value(s) for this bus interface. See C.11.

k) **vendorExtensions** (optional) holds any vendor-specific data from other namespaces, which is applicable to this bus interface. See C.10.

See also: SCR 1.4, SCR 2.14, SCR 2.15, SCR 9.4, SCR 9.5, and SCR 9.6.

6.5.1.2.1 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.

NOTE—The description of endianness is byte-centric as that is the most common least addressable unit (LAU). However, this description generally applies to any size LAU.

6.5.1.2.2 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, for example,

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.

6.5.1.3 Example

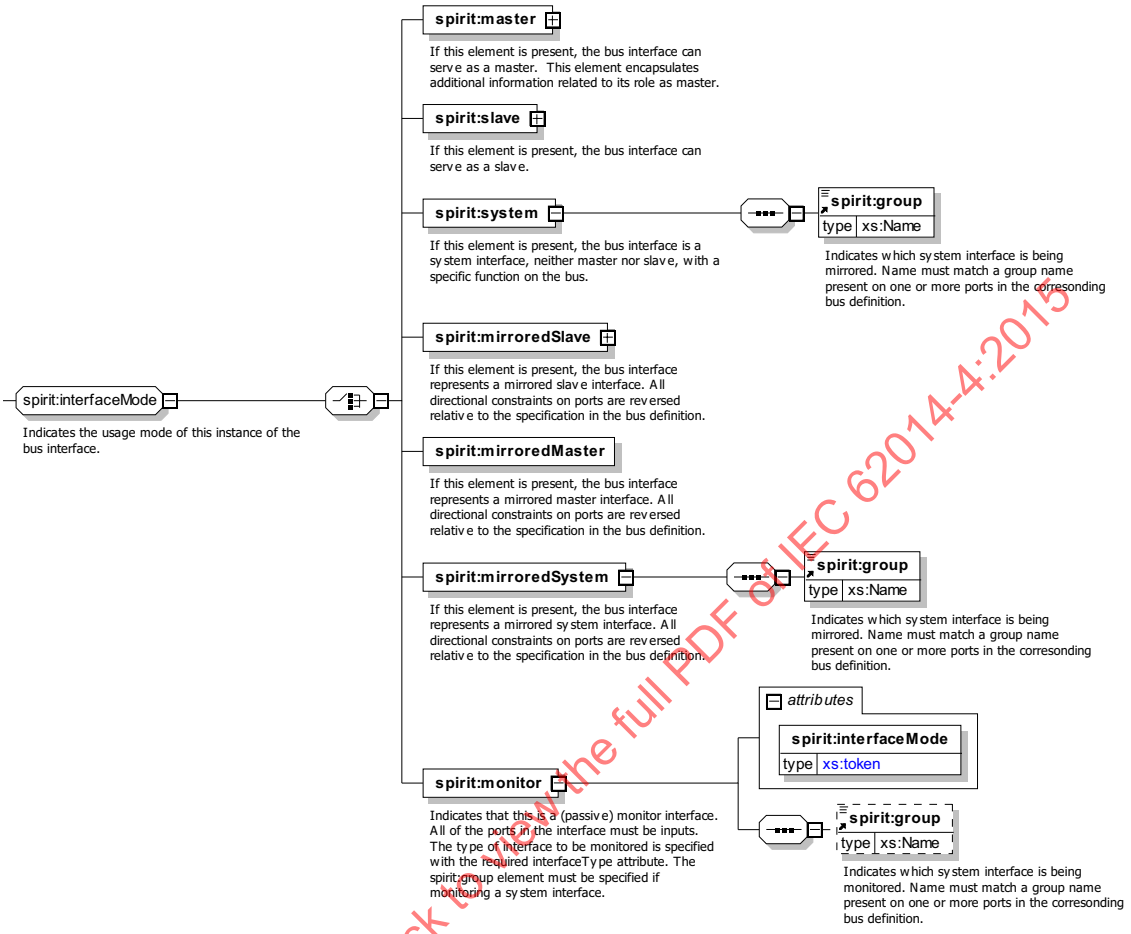
The following example shows a simple bus interface for a clock port. The interface references a bus definition and an abstraction definition.

```
<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>
```

6.5.2 Interface modes

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

6.5.2.1 Schema



6.5.2.2 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 6.5.3.

b) A **slave** interface mode (sometimes also known as a *target*) is one that responds to transactions. See 6.5.4.

c) A **system** interface mode is used for some classes of interfaces that are standard on different bus types, but do not 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***.

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 6.5.5.

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***.

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. See 6.3.3.

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***.

See also: SCR 2.12, SCR 4.3, SCR 4.4, and SCR 6.16.

6.5.2.3 Example

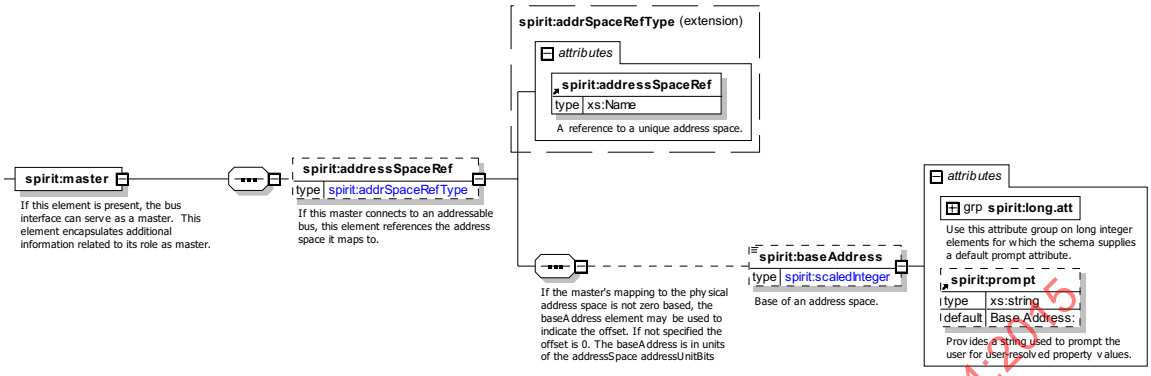
The following example shows a portion of a bus interface for an AHB bus interface. The interface mode is defined as monitor for a slave.

```
<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>
```

6.5.3 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.

6.5.3.1 Schema



6.5.3.2 Description

A **master** interface (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.

   1) **addressSpaceRef** (mandatory) attribute references a name of an address space defined in the containing description. The address space shall define the range and width for transaction on this interface. See 6.7.

   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 ***scaledInteger***, see D.15. The **baseAddress** element is configurable with attributes from ***long.att***, see C.12. The **prompt** (optional) attribute allows the setting of a string for the configuration and has a default value of **"Base Address:"**. See also: Clause 11.

See also: SCR 9.1.

6.5.3.3 Example

The following example 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`.

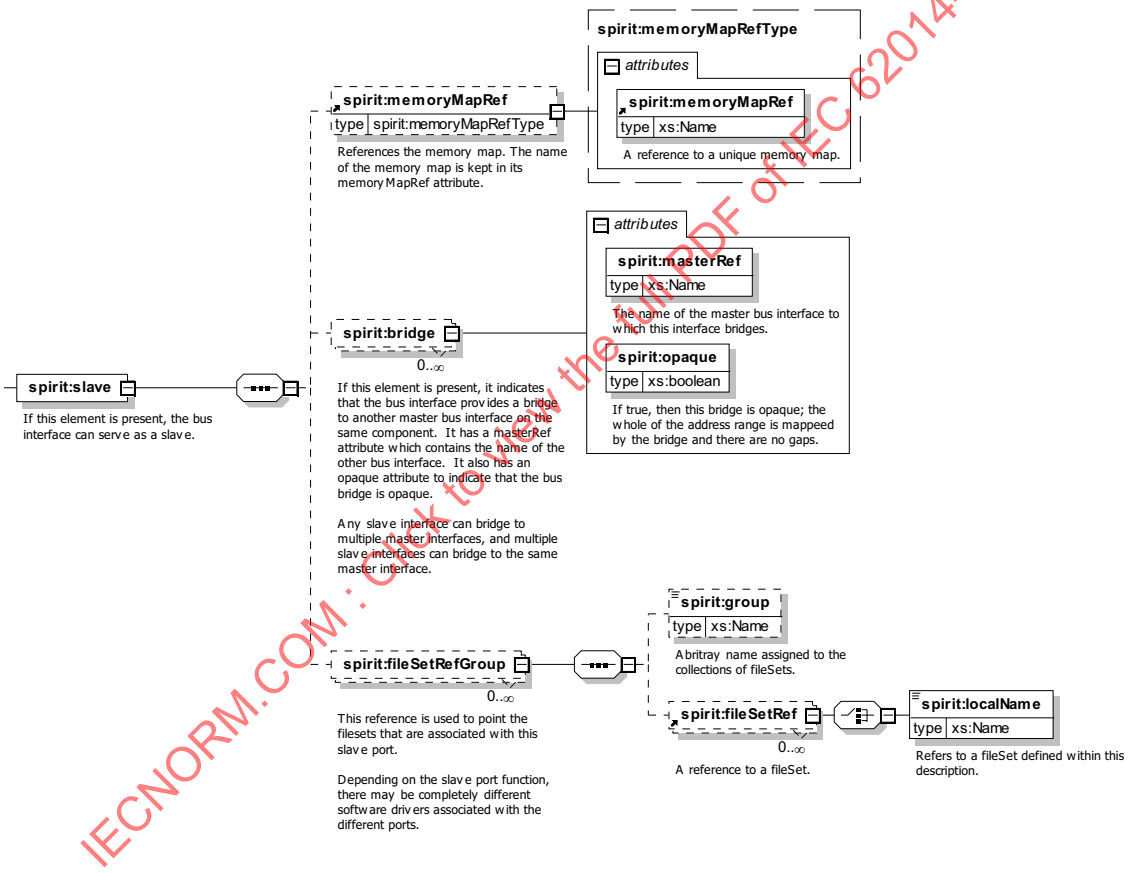
```
<spirit:busInterface>
   <spirit:name>AHBmaster</spirit:name>
   <spirit:busType spirit:vendor="amba.com" spirit:library="AMBA2"
   spirit:name="AHB" spirit:version="r2p0_5"/>
   <spirit:abstractionType spirit:vendor="amba.com" spirit:library="AMBA2"
   spirit:name="AHB_rtl" spirit:version="r2p0_5"/>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

6.5.4 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.

6.5.4.1 Schema



6.5.4.2 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 a memory map.

   The **memoryMapRef** (mandatory) attribute references a name of a memory map defined in the containing description. The memory map contains information about the range of registers, memory, or other address blocks. See 6.8.

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 (see 6.5.3) in the containing description. Under some conditions, transactions from the slave interface may be bridged to the referenced master interface, as defined by **opaque** (see also 6.4.2).

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** added and, if non-negative, the result shall exit on the subspace maps' referenced master interface's referenced address space (see 6.4.2 and Clause 11). **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 6.13). This element can further be used to reference a software driver, which can be made different for each slave port.

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

d) **fileSetRef** (optional) is an unbounded list of references to a **fileSet** by **name** within the containing document or another document referenced by the **VLNV**. See C.8.

See also: SCR 3.6 and SCR 9.2.

6.5.4.3 Example

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

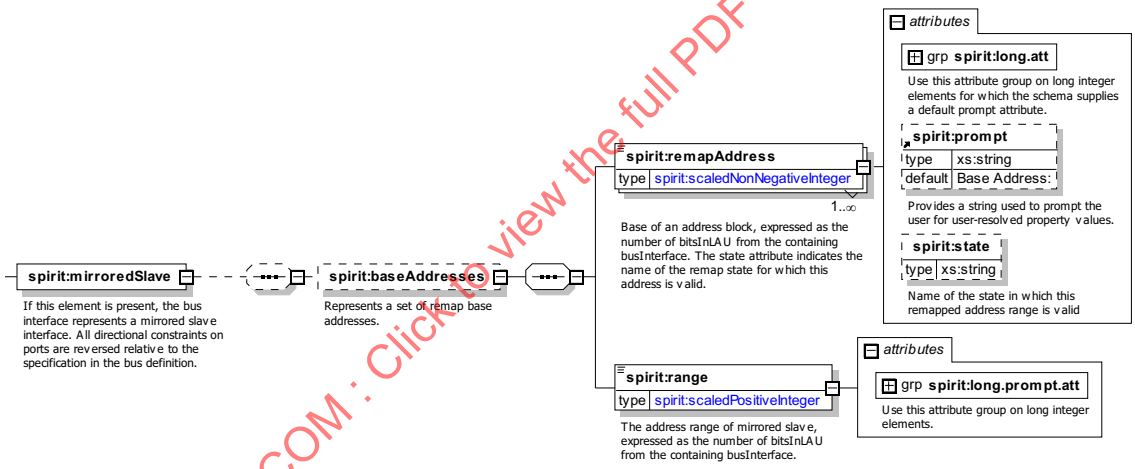
```
<spirit:busInterface>
    <spirit:name>ambaAPB</spirit:name>
    <spirit:busType spirit:vendor="amba.com" spirit:library="AMBA2"
    spirit:name="APB" spirit:version="r2p0_3"/>
    <spirit:abstractionType spirit:vendor="amba.com" spirit:library="AMBA2"
    spirit:name="APB_rtl" spirit:version="r2p0_3"/>
    <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>
```

6.5.5 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.

6.5.5.1 Schema



6.5.5.2 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 **remapAddress** is expressed as the number of addressable units based on the size of an addressable unit as defined inside the containing **busInterface**/**bitsInLau** element. The type of this element is set to ***scaledNonNegativeInteger***, see D.15. The **remapAddress** element is configurable with attributes from ***long.att***, see C.12. 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 **remapState**/**name** for which the **remapAddress** and **range** apply. See 6.9.2.

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 as defined inside the containing **busInterface**/**bitsInLau** element. See 6.5.1. The type of this element is set to

scaledPositiveInteger. The **range** element is configurable with attributes from ***long.prompt.att***, see C.12.

6.5.5.3 Example

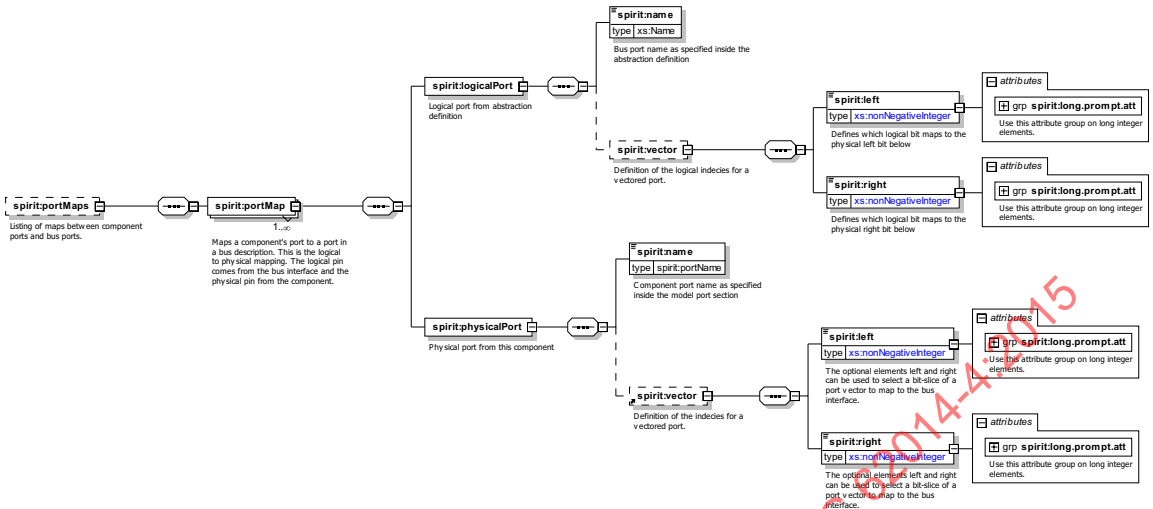
This example shows a portion of a bus interface for an AHB mirroredSlave bus interface. The interface contains two remap addresses. The first does not have a state attribute and is always active unless a named state is active; in this case, the base address of the connected slave is offset by `0x00000000`.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

6.5.6 Port map

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

6.5.6.1 Schema



6.5.6.2 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 C.12.

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 C.12.

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 6.3.4.1.

See also: SCR 6.1, SCR 6.2, SCR 6.3, SCR 6.4, SCR 6.5, SCR 6.6, SCR 6.7, SCR 6.12, SCR 6.13, SCR 6.19, SCR 6.20, SCR 6.21, SCR 6.22, SCR 6.23, SCR 6.24, and SCR 6.25.

6.5.6.3 Example

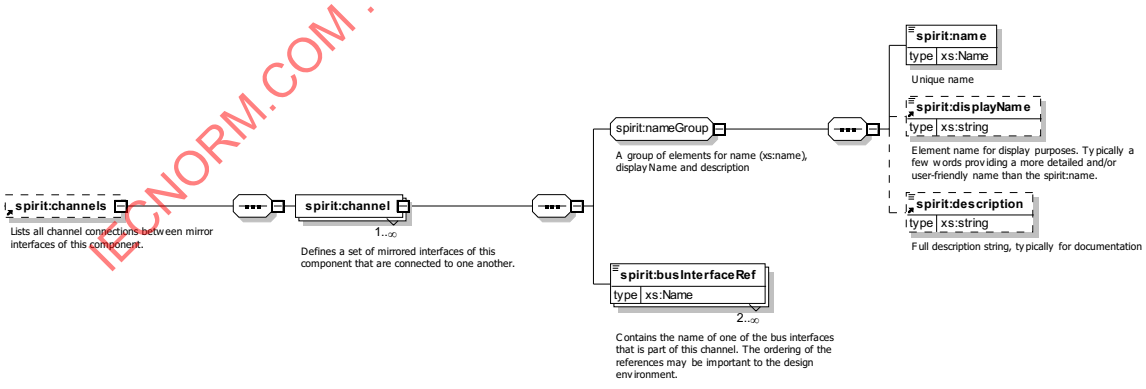
The following example shows a portion of a bus interface for an APB bus interface. The logical port `PADDR` is mapped to the lower 12 bits of the physical port `paddr`, and the logical port `PWRITE` is mapped to the physical port `pwrite`.

```
<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>
```

6.6 Component channels

6.6.1 Schema

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



6.6.2 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 is defined in C.1. The **name** element shall be unique within the containing **channels** element.

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 DE in some way and shall be maintained. The **busInterfaceRef** element is of type ***Name***. See 6.5.1.

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

6.6.3 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:mirroredMaster>...</spirit:mirroredMaster>
  </spirit:busInterface>
  <spirit:busInterface>
    <spirit:name>InterfaceB</spirit:name>
    <spirit:busType>...</spirit:busType>
    <spirit:mirroredSlave>...</spirit:mirroredSlave>
  </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>
```

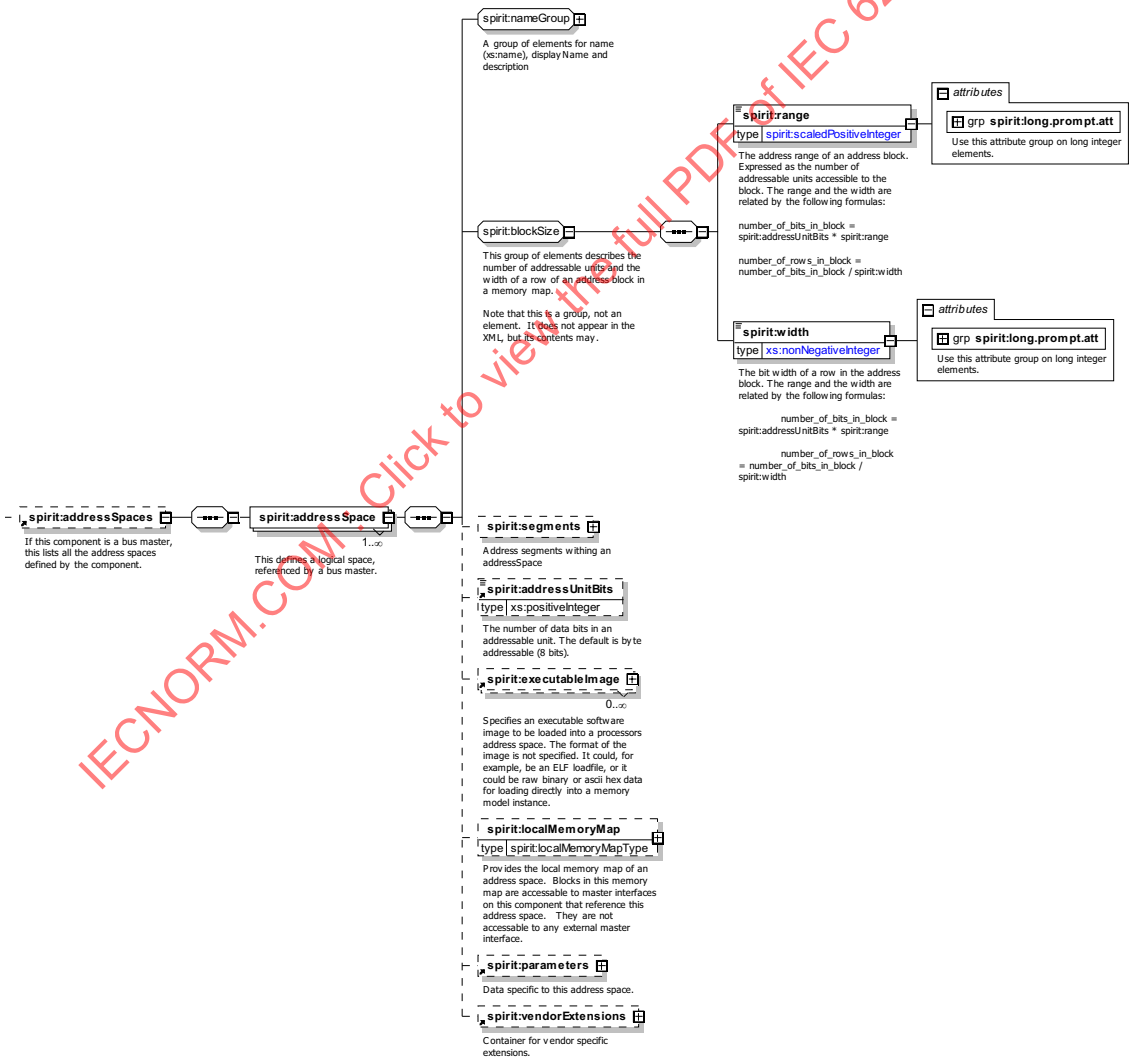
6.7 Address spaces

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 interface. Some components may have one address space 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 associated with multiple master interfaces—one for instruction and the other for data.

6.7.1 addressSpaces

6.7.1.1 Schema

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



6.7.1.2 Description

The **addressSpaces** element contains an unbounded 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 is defined in C.1. The **name** element shall be unique within the containing **addressSpaces** element.

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

1) **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 C.12.

2) **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 C.12.

c) **segments** (optional) describes a portion of the address space starting at an address offset and continuing for a given range. A **segment** can be referenced by a **subspaceMap**. See 6.7.7.

d) **addressUnitBits** (optional) defines the number of data bits in each address increment of the address space. If this element is not present, it is presumed to be `8`.

e) **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. See 6.7.3.

f) **localMemoryMap** (optional) describes a local memory map that is seen exclusively by this master bus interface viewing this address space. See 6.7.7.

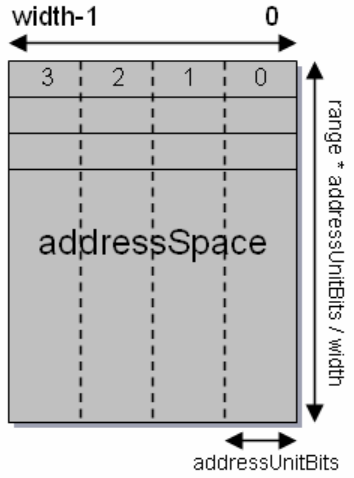
g) **parameters** (optional) specifies any parameter data value(s) for this address space. See C.11.

h) **vendorExtensions** (optional) holds any vendor-specific data from other namespaces, which is applicable to this address space. See C.10.

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 9.3 and SCR 9.8.

6.7.1.3 Example

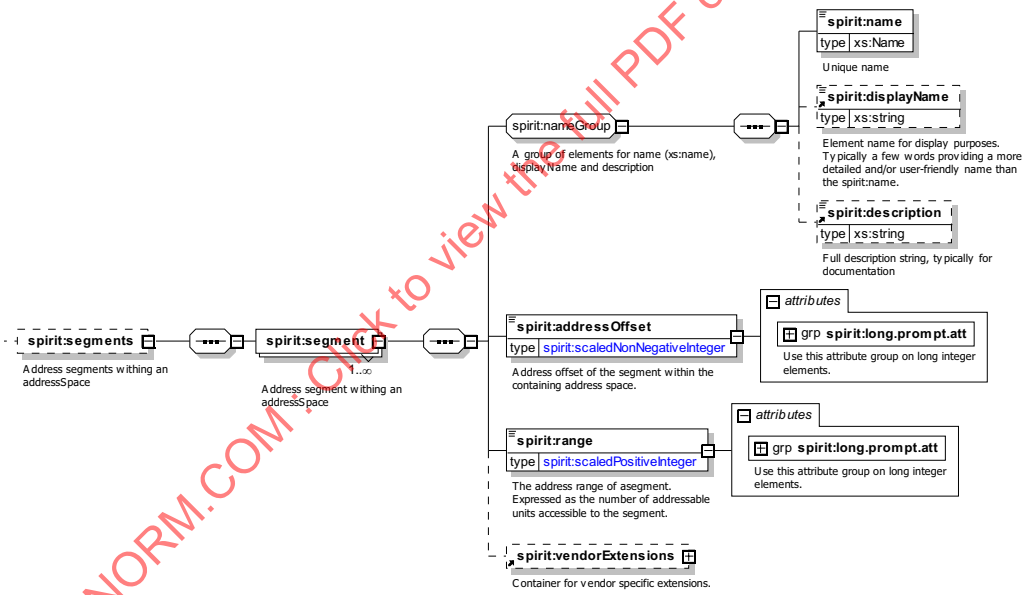
The following example shows the definition of an address space with a range (length) of 4 GB and a width of 32 bits.

```
<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>
```

6.7.2 Segments

6.7.2.1 Schema

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



6.7.2.2 Description

The **segments** element contains an unbounded list of **segment** elements. Each **segment** describes the location and size of an area in the containing **addressSpace**.The **segment** element contains the following elements.

a) ***nameGroup*** group is defined in C.1. The **name** element shall be unique within the containing **segments** element.

b) **addressOffset** (mandatory) describes, in addressing units from the containing **addressSpace/ addressUnitBits** element, the offset from the start of the **addressSpace**. The **addressOffset** element is of type ***scaledNonNegativeInteger***. The **addressOffset** element is configurable with attributes from ***long.prompt.att***, see C.12.

c) **range** (mandatory) gives the address range of an address space segment. This is expressed as the number of addressable units of the address space segment. 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 C.12.

d) **vendorExtensions** (optional) holds any vendor-specific data from other namespaces, which is applicable to this address space. See C.10.

See also: SCR 9.8.

6.7.2.3 Example

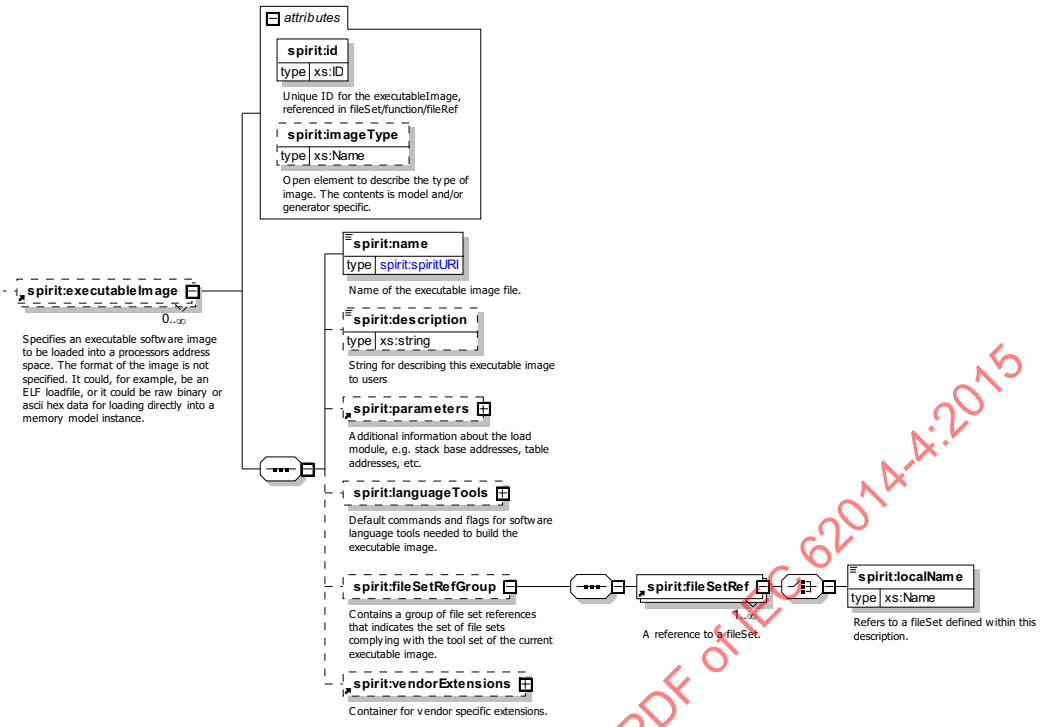
The following example shows the definition of an address space with a range (length) of 4 GB and a width of 32 bits. The address space contains two segments, one starting at `0x10000000` with a range of 32 MB, the second segment starts at `0x80000000` with a range of 1 GB.

```
<spirit:addressSpaces>
    <spirit:addressSpace>
        <spirit:name>main</spirit:name>
        <spirit:range>4G</spirit:range>
        <spirit:width>32</spirit:width>
        <spirit:segments>
            <spirit:segment>
                <spirit:name>segment1</spirit:name>
                <spirit:addressOffset>0x10000000</spirit:addressOffset>
                <spirit:range>32M</spirit:range>
            </spirit:segment>
            <spirit:segment>
                <spirit:name>segment2</spirit:name>
                <spirit:addressOffset>0x80000000</spirit:addressOffset>
                <spirit:range>1G</spirit:range>
            </spirit:segment>
        </spirit:segments>
        <spirit:addressUnitBits>8</spirit:addressUnitBits>
    </spirit:addressSpace>
</spirit:addressSpaces>
```

6.7.3 executableImage

6.7.3.1 Schema

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



6.7.3.2 Description

The **executableImage** element 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 and contains the following elements.

a) **id** (mandatory) attribute uniquely identifies the **executableImage** for reference in a **fileSet/function/fileRef**. The **id** attribute is of type ***ID***.

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

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

d) **description** (optional) allows a textual description of the address space. The **description** element is of type ***string***.

e) **parameters** (optional) specifies any parameter data value(s) for this executable object. See C.11.

f) **languageTools** (optional) contains further elements to describe the information needed to build the executable image. See 6.7.4.

g) **fileSetRefGroup** (optional) element contains a list of **fileSetRef** subelements, each one containing the name of a file set associated with this **executableImage**. See 6.13.

h) **vendorExtensions** (optional) holds any vendor-specific data from other namespaces, which is applicable to this address space. See C.10.

See also: SCR 9.3.

6.7.3.3 Example

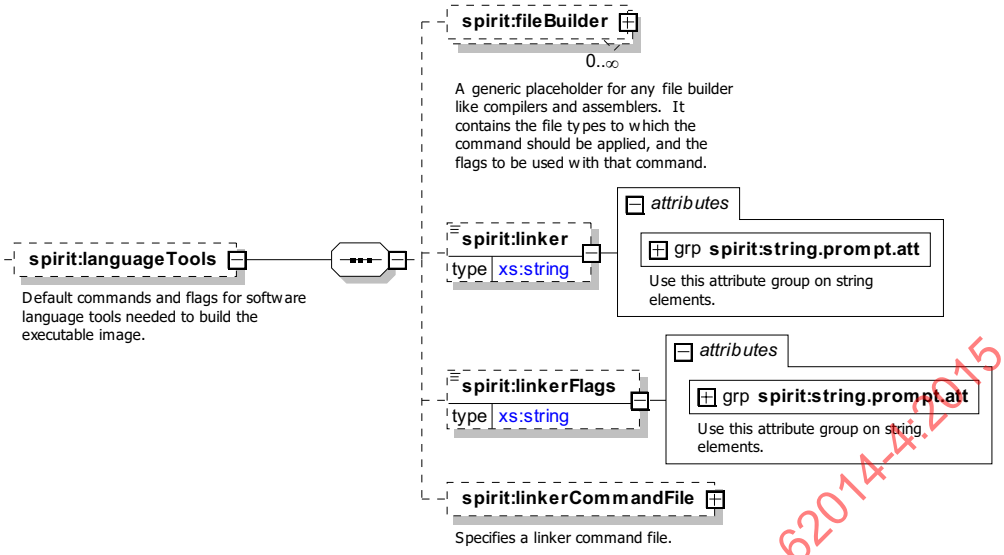
The following example shows the definition of a binary executable produced using the Gnu C Compiler (GCC) software tools.

```
<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>
         <spirit:localName>calculatorAppC</spirit:localName>
      </spirit:fileSetRef>
      <spirit:fileSetRef>
         <spirit:localName>mathFunctions</spirit:localName>
      </spirit:fileSetRef>
      <spirit:fileSetRef>
         <spirit:localName>coreLib-gnu</spirit:localName>
      </spirit:fileSetRef>
   </spirit:fileSetRefGroup>
</spirit:executableImage>
```


6.7.4 languageTools

6.7.4.1 Schema

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



6.7.4.2 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 6.7.5.

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 C.12.

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 C.12.

d) **linkerCommandFile** (optional) documents a file containing commands the linker follows. See 6.7.6.

See also: SCR 9.7.

6.7.4.3 Example

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

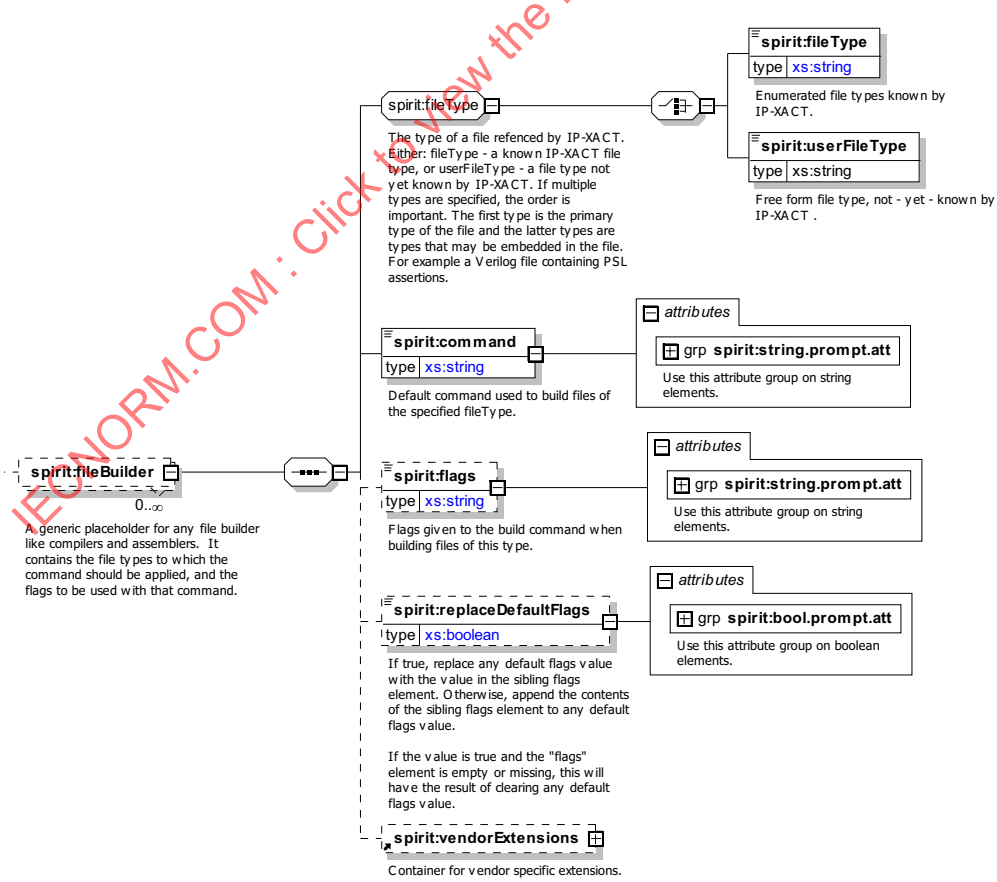
```
<spirit:languageTools>
   <spirit:fileBuilder>
      <spirit:fileType>cSource</spirit:fileType>
      <spirit:command spirit:id="gccCompilerDefault"> gcc</spirit:command>
      <spirit:flags spirit:id="gccCFlags">-c -g -I${INCLUDES_LOCATION}/
   software/include -I${GCC_LIBRARY}/common/include</spirit:flags>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

6.7.5 fileBuilder

6.7.5.1 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.



6.7.5.2 Description

The **fileBuilder** element contains the following elements.

a) ***fileType*** (mandatory) group contains one or more of the elements defined in C.9.

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 C.12.

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***, see C.12.

d) **replaceDefaultFlags** (optional) documents, when **true**, flags that replace any of the default flags from a build script generator. If **false**, the flags contained in the flags element are appended to the current command. If the value is **true** and the flags element is empty or does not exist, this has the effect of clearing all the flags in build script generator. The **replaceDefaultFlags** element is of type ***boolean***. The **replaceDefaultFlags** element is configurable with attributes from ***bool.prompt.att***, see C.12.

e) **vendorExtensions** (optional) holds vendor-specific data from other namespaces applicable to building this software source code file into an executable object file. See C.10.

6.7.5.3 Example

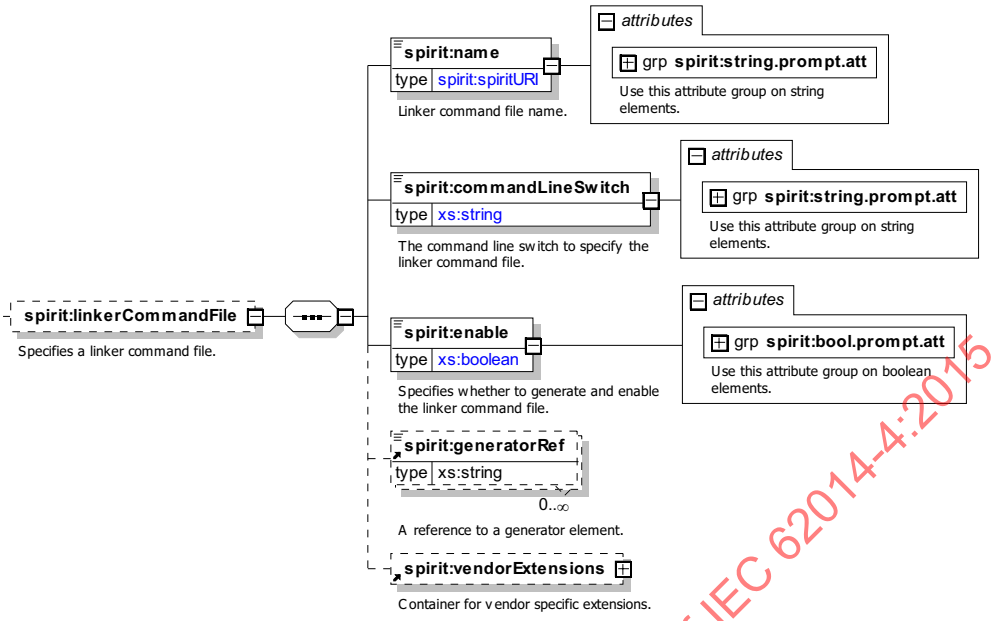
The following example shows the specification for compiling a C language file using GCC.

```
<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>
```

6.7.6 linkerCommandFile

6.7.6.1 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.



6.7.6.2 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 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 C.12.

b) **commandLineSwitch** (mandatory) documents the flag on the command line specifying the linker command file. The **commandLineSwitch** element is of type ***spiritURI***. The **commandLineSwitch** element is configurable with attributes from ***string.prompt.att,*** see C.12.

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 C.12. The following also apply. For:

1) **enable=true** and a **generatorRef**, run the generator to link the **executableImage**; it may use the other elements to link the **executableImage**.

2) **enable=true** and no **generatorRef**, run the linker with the **-commandLineSwitch** *name* (the command file).

3) **enable=false**, run the linker with **linkerFlags**.

d) **generatorRef** (optional) references the generator (in the containing component) that creates and launches the linker command. There may be any number of these elements present. The **generator-Ref** element is of type ***string***. See 6.12.

e) **vendorExtensions** (optional) holds any vendor-specific data from other namespaces applicable to using this linker. See C.10.

6.7.6.3 Example

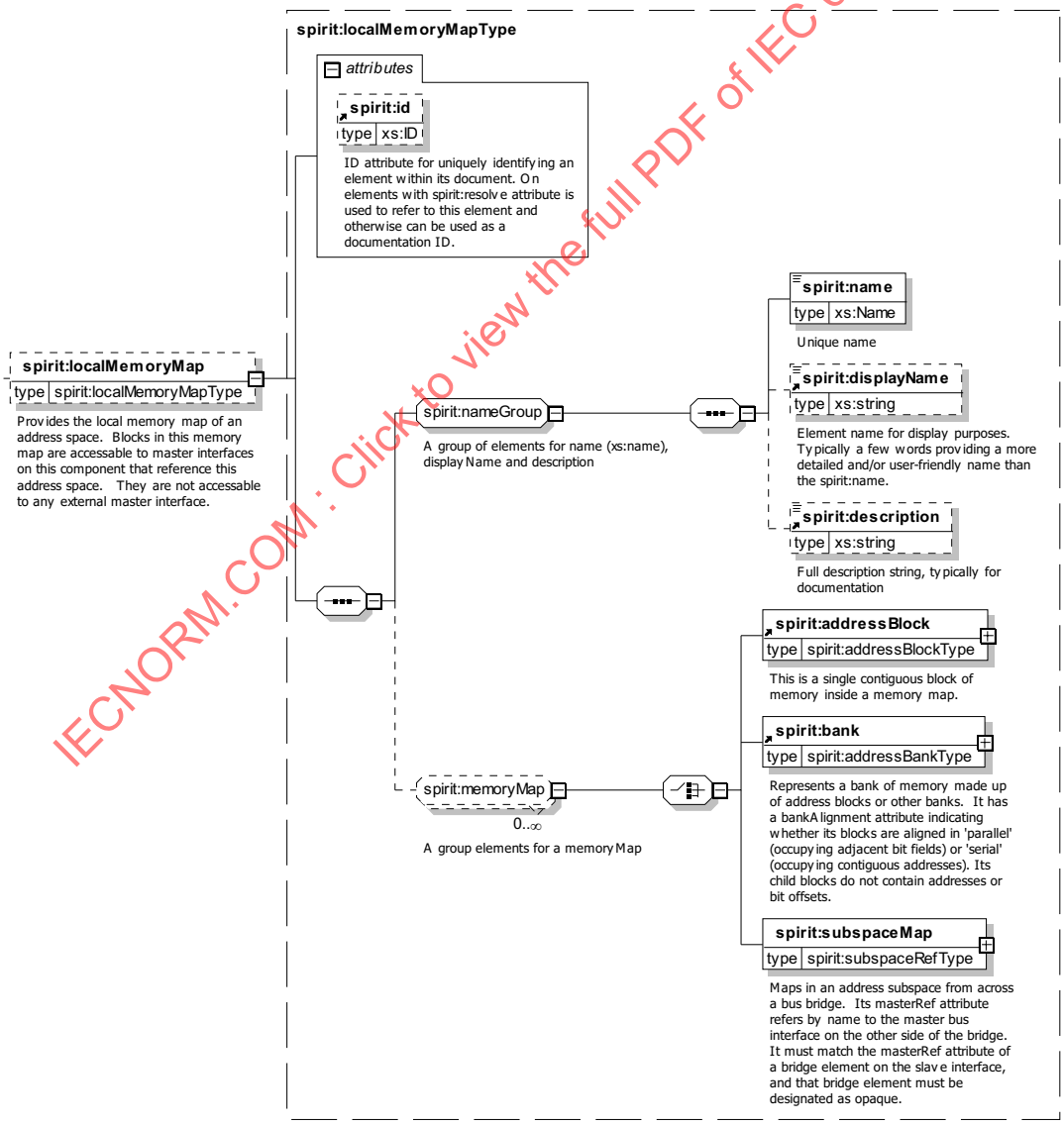
The following example shows the definition of a status register which can be accessed within a component during verification.

```
<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>
```

6.7.7 Local memory map

6.7.7.1 Schema

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



6.7.7.2 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. If the master interface containing a local memory map is bridged from a slave interface (see 6.4.2), the local memory map is visible from this slave interface. The **localMemoryMap** element contains an **id** (optional) attribute that assigns a unique identifier to the containing element for reference throughout the containing description. **localMemoryMap** contains the following mandatory and optional elements.

a) ***nameGroup*** group is describe in C.1.

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

1) **addressBlock** describes a single block. See 6.8.2.

2) **bank** represents a collection of address blocks, banks, or subspace maps. See 6.8.5.

3) **subspaceMap** maps the address subspaces of master interfaces into the slave's memory map. See 6.8.9.

6.7.7.3 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.

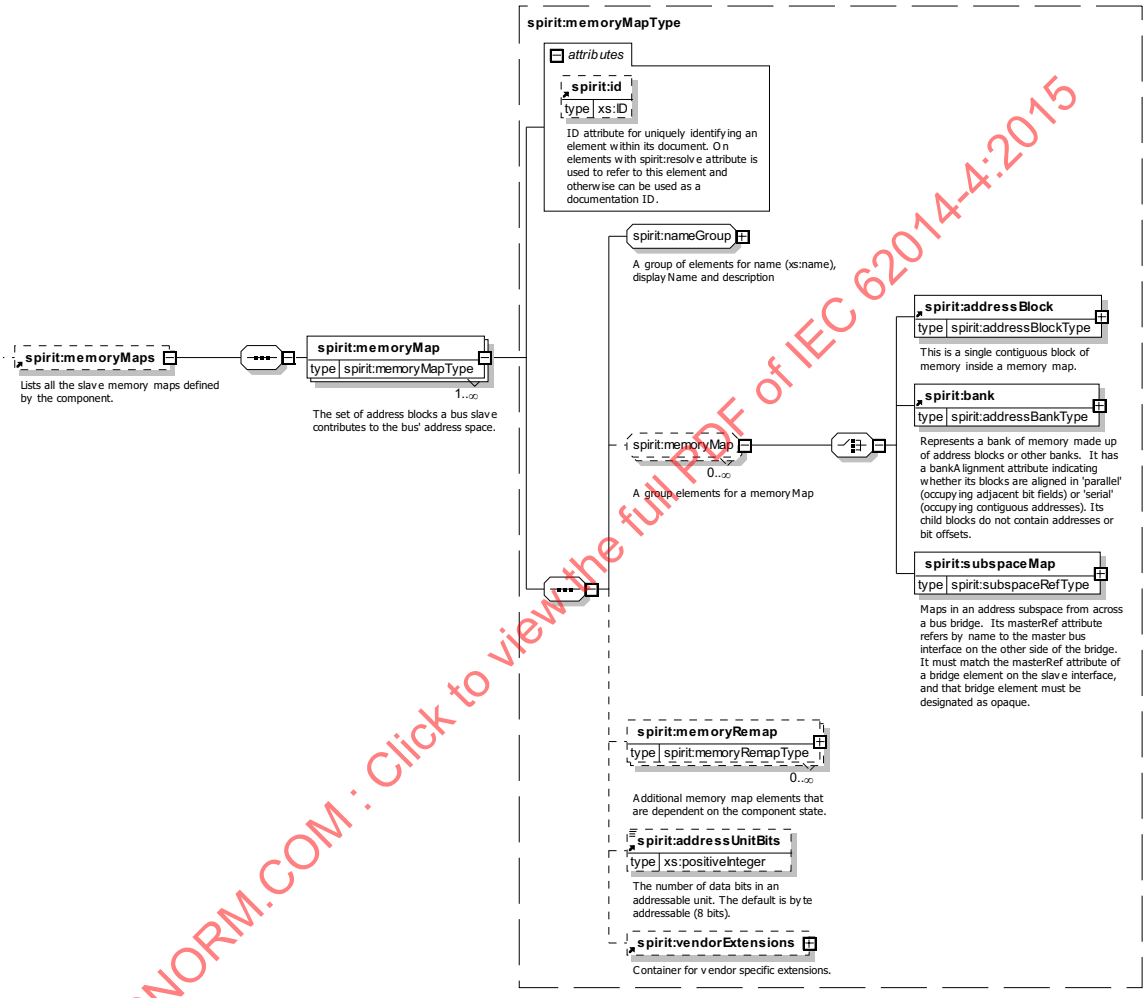
```
<spirit:localMemoryMap>
    <spirit:name>secureRegs</spirit:name>
    <spirit:displayName>Secure Registers</spirit:displayName>
    <spirit:description>Secure registers area</spirit: description>
    <spirit:addressBlock>
        <spirit:baseAddress spirit:id="secureRegs">0x50000000</
    spirit:baseAddress>
        <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>
```

6.8 Memory maps

6.8.1 memoryMaps

6.8.1.1 Schema

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



6.8.1.2 Description

A memory map can be defined for each slave interface of a component. The **memoryMaps** element contains an unbounded list of **memoryMap** elements. The **memoryMap** elements are referenced by the component's slave interface. The **memoryMap** element contains an **id** (optional) attribute that assigns a unique identifier to the containing element for reference throughout the containing description. **memoryMap** contains the following mandatory and optional elements.

a) ***nameGroup*** group is defined in C.1. The **name** element shall be unique within the containing **memoryMaps** element.

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

1) **addressBlock** describes a single block. See 6.8.2.

2) **bank** represents a collections of address blocks, banks, or subspace maps. See 6.8.5.

3) **subspaceMap** maps the address subspaces of master interfaces into the slave's memory map. See 6.8.9.

c) The optional **memoryRemap** element describes additional address blocks, banks, and subspace maps of a slave bus interface in a specific remap **state**.

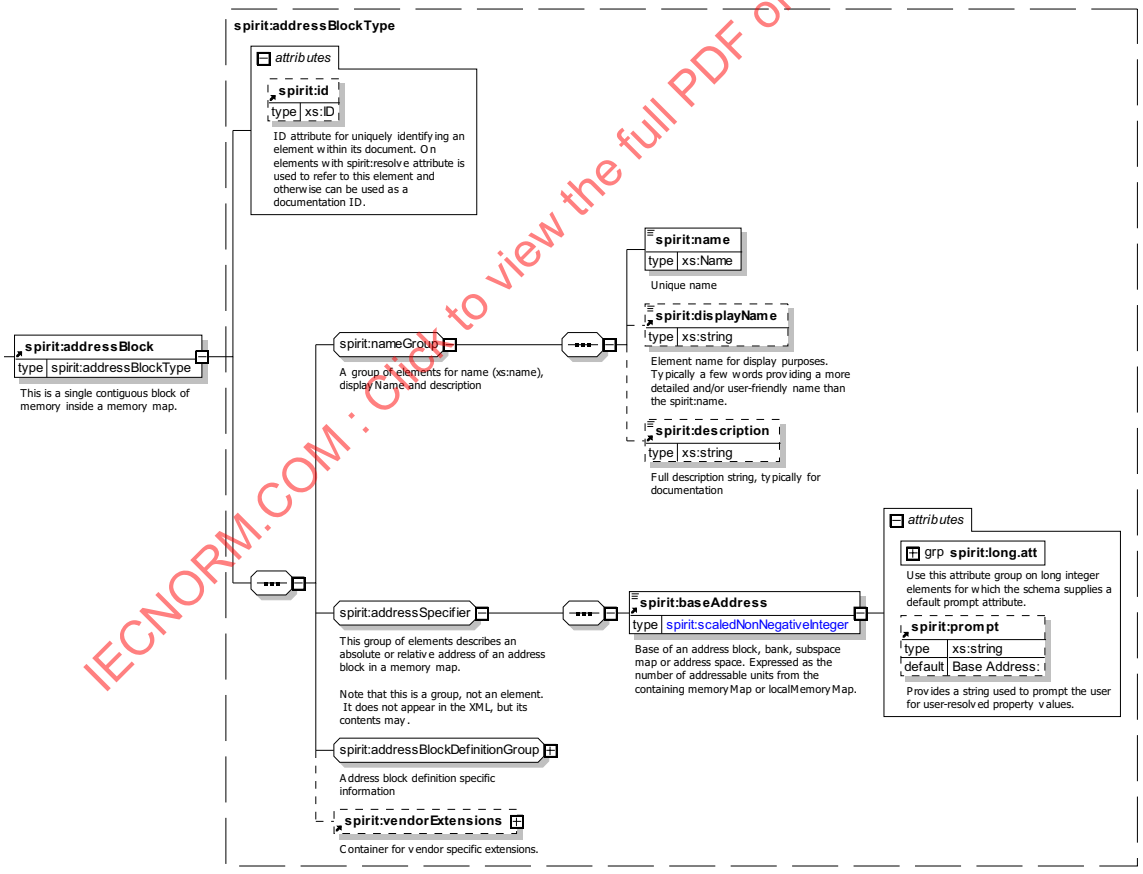
d) The optional **addressUnitBits** element defines the number of data bits in each address increment of the memory map. This is required to allow the elements in the memory map to define items such as register offsets. The **addressUnitBits** element is of type ***positiveInteger***.

e) **vendorExtensions** (optional) adds any extra vendor-specific data related to the memory map. See C.10.

6.8.2 Address block

6.8.2.1 Schema

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



6.8.2.2 Description

The **addressBlock** element describes a single, contiguous block of memory that is part of a memory map. The **addressBlock** element contains an **id** (optional) attribute that assigns a unique identifier to the containing element for reference throughout the containing description. **addressBlock** contains the following mandatory and optional elements.

a) ***nameGroup*** is defined in C.1. The **name** of the **addressBlock**, **subspaceMap**, **bank**, and **memoryRemap** shall be unique within the containing **memoryMap**, **localMemoryMap**, or **memoryRemap** element.

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

baseAddress (mandatory) specifies the starting address of the block. The **baseAddress** is expressed in addressing units from the containing **memoryMap/addressUnitBits** or **localMemoryMap/addressUnitBits** element. The **baseAddress** element is of type ***scaledNonNegativeInteger***. The **baseAddress** element is configurable with attributes from ***long.att***, see C.12. The **prompt** (optional) attribute allows the setting of a string for the configuration and has a default value of **"Base Address:"**.

c) ***addressBlockDefinitionGroup*** group contains definition information about address blocks. See 6.8.3.

d) **vendorExtensions** (optional) adds any extra vendor-specific data related to the address block. See C.10.

See also: SCR 8.1 and SCR 8.16.

6.8.2.3 Example

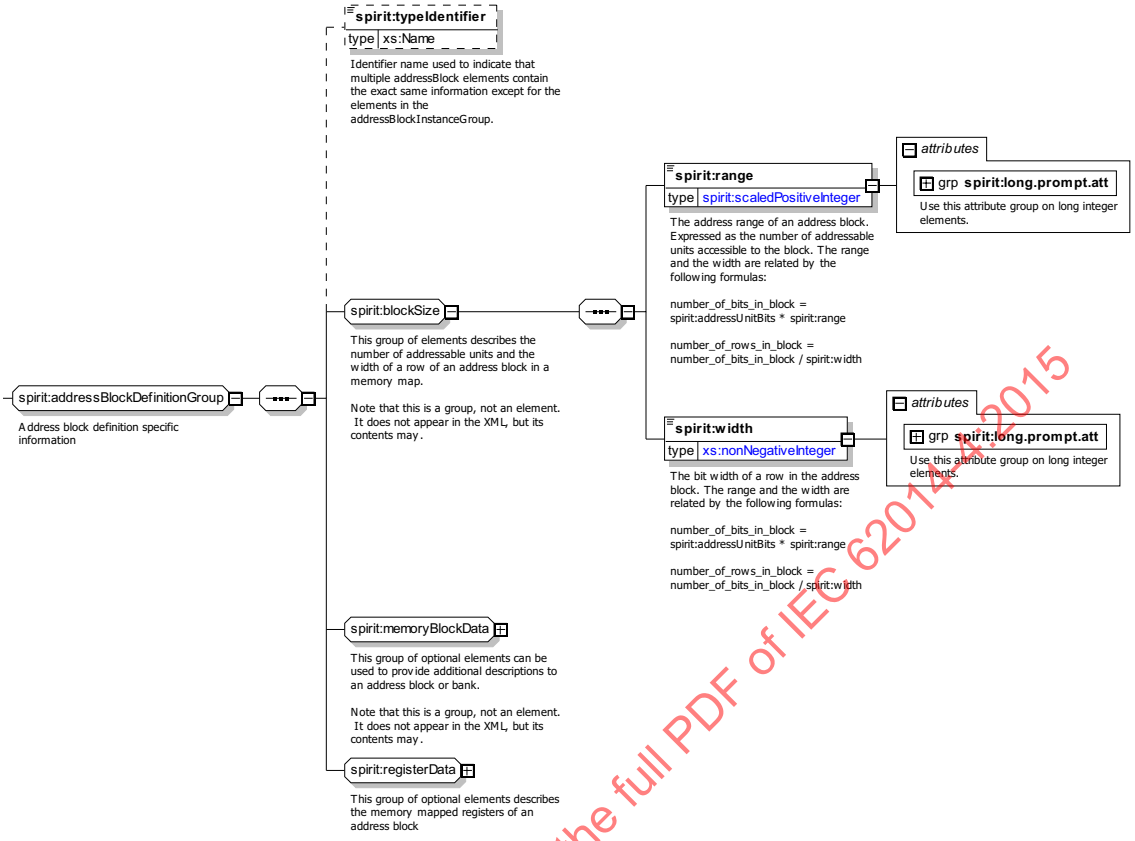
The following example shows an address block starting at address `0x1000` in memory map `map1`, containing 64 addressable 8-bit units, organized into larger 32-bit units.

```
<spirit:memoryMap>
    <spirit:name>map1</spirit:name>
    <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>
```

6.8.3 Address block definition group

6.8.3.1 Schema

The following schema details the information contained in the ***addressBlockDefinitionGroup*** group, which may appear in an **addressBlock** element.



6.8.3.2 Description

The ***addressBlockDefinitionGroup*** group describes the definition information about address blocks. It contains the following mandatory and optional elements.

a) **typeIdentifier** (optional) indicates multiple address block elements with the same **typeIdentifier** in the same description contain the exact same information for the elements in the **addressBlockDefinitionsGroup**. The **typeIdentifier** element is of type ***Name***.

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

1) **range** (mandatory) gives the address range of an address block. This is expressed as the number of addressable units. The size of an addressable unit is defined inside the containing **memoryMap/addressUnitBits** or **memoryMap/addressUnitBits** element. The **range** element is of type ***scaledPositiveInteger***. The **range** element is configurable with attributes from ***long.prompt.att***, see C.12.

2) **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 C.12.

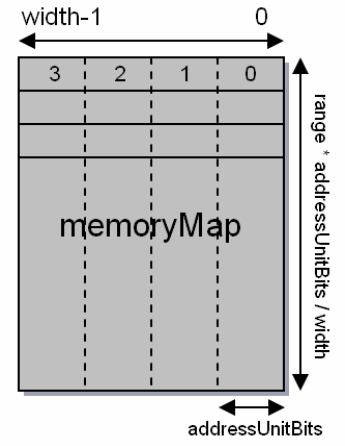
c) ***memoryBlockData*** group contains information about usage, access, volatility, and other parameters. See 6.8.4.

d) ***registerData*** group contains information about the grouping of bits into registers and fields. See 6.10.1.

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 and SCR 7.15.

6.8.3.3 Example

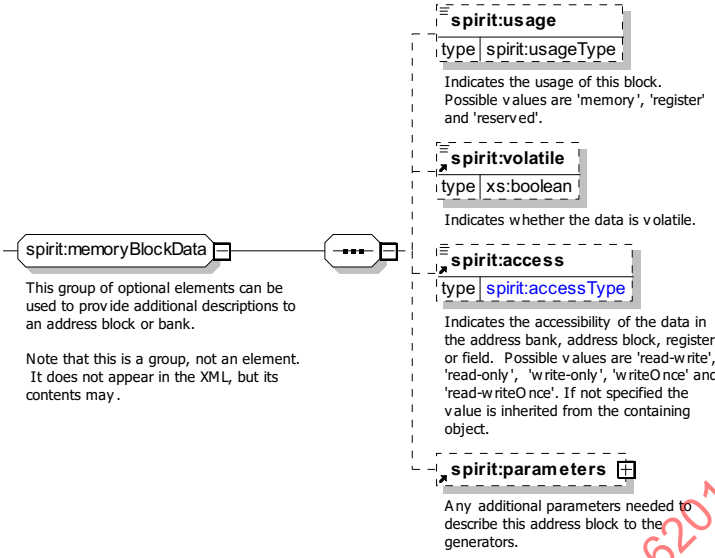
The following example shows an address block starting at address `0` in memory map `map1`, containing 1024 addressable 8-bit units, organized into larger 32-bit units.

```
<spirit:memoryMap>
    <spirit:name>map1</spirit:name>
    <spirit:addressBlock>
        <spirit:name>AB1</spirit:name>
        <spirit:baseAddress>0</spirit:baseAddress>
        <spirit:range>1K</spirit:range>
        <spirit:width>32</spirit:width>
    </spirit:addressBlock>
    <spirit:addressUnitBits>8</spirit:addressUnitBits>
</spirit:memoryMap>
```

6.8.4 memoryBlockData group

6.8.4.1 Schema

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



6.8.4.2 Description

The ***memoryBlockData*** group is a collection of elements that contains further specification of **addressBlock** or **bank** elements. It contains the following elements.

a) **usage** (optional) specifies the type of usage for the address block or bank to which it belongs.

   1) For an **addressBlock**:

      i) **memory** defines, when the **access** element is set to **read-only**, the entire range of the **addressBlock** as a ROM. If the **access** element is set to **read-write**, the entire range of the **addressBlock** is a RAM. If the **access** element is set to **write-only**, the entire range of the **addressBlock** is a write-only memory. This usage type shall not contain registers.

      ii) **register** defines the entire range of the **addressBlock** as possible locations for registers.

      iii) **reserved** defines the entire range of the **addressBlock** as reserved or for unknown usage to IP-XACT. This type shall not contain registers.

      iv) If unspecified, the presumed value for **usage** shall be **register** if the **addressBlock** contains **register** elements; otherwise it is **reserved**.

   2) For a **bank**:

      i) **memory** defines all containing **addressBlock** elements are of this access type.

      ii) **register** defines all containing **addressBlock** elements are of this access type.

      iii) **reserved** defines all containing **addressBlock** elements are of this access type.

      iv) Unspecified usage means the bank may contain a mixture of **memory**, **register**, and **reserved addressBlock** elements.

b) **volatile** (optional) when **true** indicates the case of a write followed by read, or in the case of two consecutive reads, there is no guarantee what is returned by the read on the second transaction or that this return value is consistent with the write or read of the first transaction. The element implies there is some additional mechanism by which these registers can acquire new values other than reads/writes/resets and other access methods known to IP-XACT. If this element is not present, it is presumed to be **false** for a **field** and unspecified for **bank**, **addressBlock**, or **register**. The **volatile** element is of type ***boolean***.

c) **access** (optional) specifies the accessibility of the data in the address block. If the **usage** element is **reserved**, this element has no meaning. If the **access** is not specified, the value shall be inherited from the containing **bank** or default to **read-write** if this element is contained in a **memoryMap**.

   i) **read-write** defines, when the **usage** element is **memory**, the entire range is a RAM. If the **usage** element is **register**, then any access type for a register or alternate register is allowed.

   ii) **read-only** defines, when the **usage** element is **memory**, the entire range is a ROM. If the **usage** element is **register**, then an access type shall be **read-only** for a register or alternate register.

   iii) **write-only** defines, when the **usage** element is **memory**, the entire range is a write-only memory. If the **usage** element is **register**, then an access type shall be **write-only** or **writeOnce** for a register or alternate register.

   iv) **read-writeOnce** defines, when the **usage** element is **memory**, the entire range is a RAM that is writable once after power up. If the **usage** element is **register**, then the access type for a register or alternate register shall be **read-only**, **read-writeOnce**, **write-only**, or **writeOnce**.

   v) **writeOnce** defines, when the **usage** element is **memory**, the entire range is a write-only memory that is writable once after power up. If the **usage** element is **register**, then the access type for a register or alternate register shall be **writeOnce**.

d) **parameters** (optional) details any additional parameters that describe the address block for generator usage. See C.11.

See also: SCR 8.3, SCR 8.4, SCR 8.6, SCR 8.7, SCR 8.9, SCR 8.10, SCR 8.11, SCR 8.13, and SCR 8.14.

6.8.4.3 Example

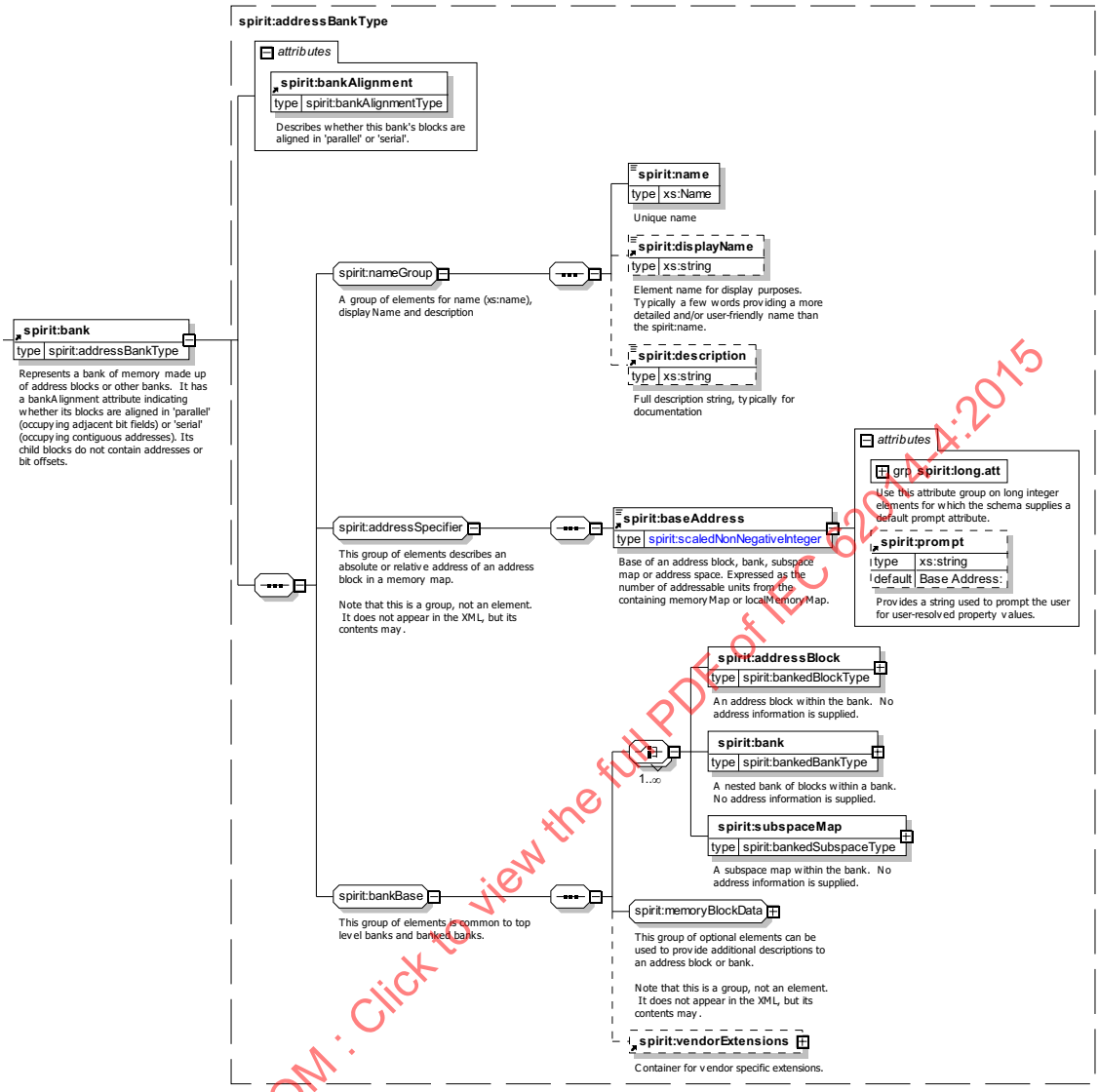
The following example shows an address block starting at address `0x0` containing 64 addressable memory locations of 8 bits, organized into larger 32-bit units.

```
<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>
```

6.8.5 Bank

6.8.5.1 Schema

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



6.8.5.2 Description

The **bank** element allows multiple **addressBlocks**, **banks**, or **subspaceMaps** to be concatenated together horizontally or vertically as 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 assigned an address in sequence.

b) ***nameGroup*** is defined in C.1. The **name** of the **addressBlock**, **subspaceMap**, **bank**, and **memoryRemap** shall be unique within the containing **memoryMap**, **localMemoryMap**, or **memoryRemap** element.

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

baseAddress (mandatory) specifies the starting address of the block. The **baseAddress** is expressed in addressing units from the containing **memoryMap/addressUnitBits** or **localMemoryMap/addressUnitBits** element. The type of this element is set to ***scaledNonNegativeInteger***. The **baseAddress** element is configurable with attributes from ***bool.prompt.att***, see C.12. The **prompt** attribute allows the setting of a string for the configuration and has a default value of **"Base Address:".**

d) ***bankBase*** group includes 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 6.8.6.
2) **bank** (multiple usage allowed) is a bank within the bank. This allows for complex configurations with nested banks. See 6.8.7.
3) **subspaceMap** (multiple usage allowed) is a reference to the master's address map for inclusion in the bank. See 6.8.9.
4) ***memoryBlockData*** group contains information about usage, access, volatility, and other parameters. See 6.8.4.
5) **vendorExtensions** adds any extra vendor-specific data related to this bank. See C.10.

See also: SCR 8.2.

6.8.5.3 Example

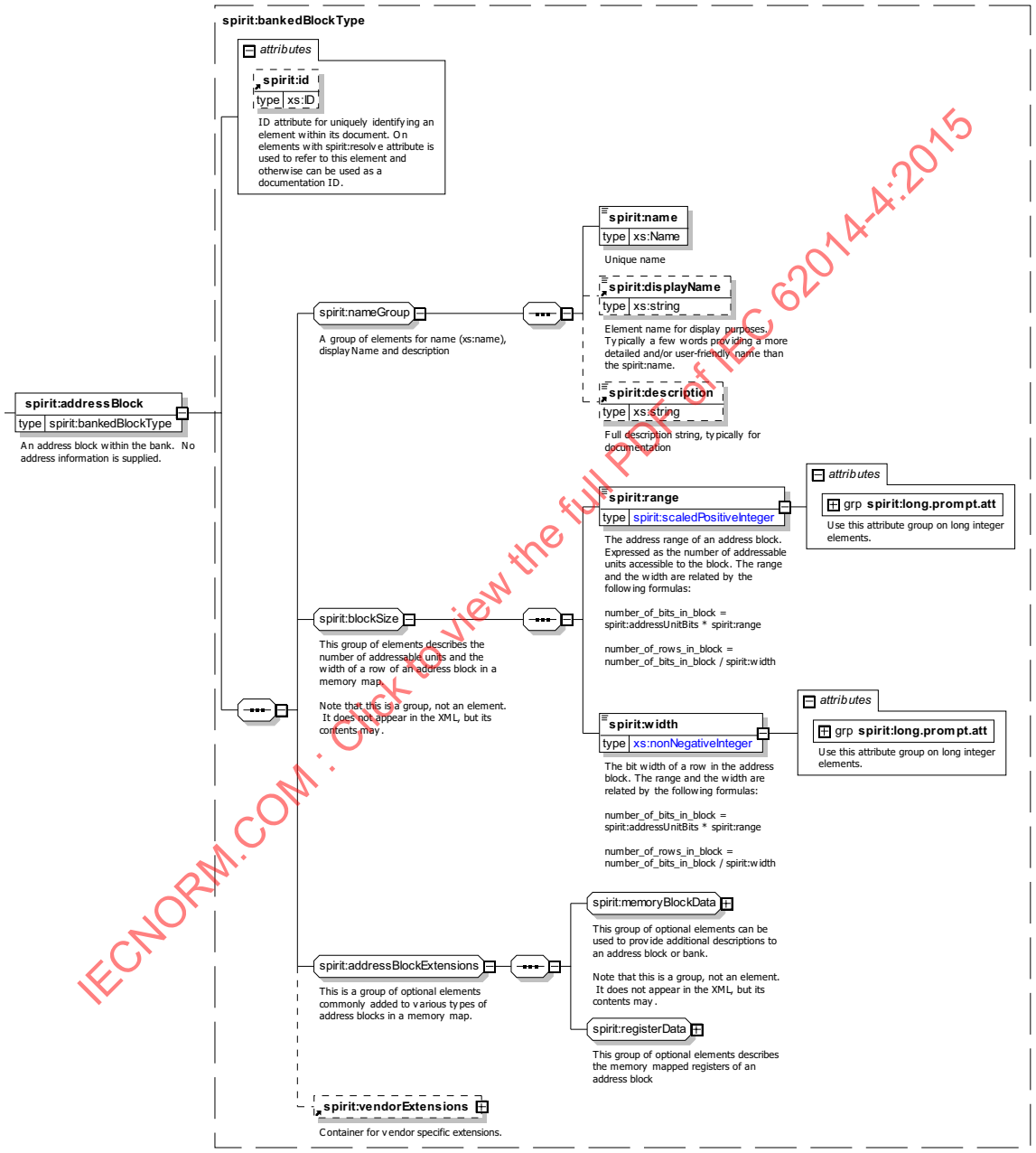
The following example shows a serial bank with four memory blocks of 1 k units of 32-bit data. The only address specified is `0x10000`, but this causes address block `ram0`, `ram1`, `ram2`, and `ram3` to be mapped to addresses `0x10000`, `0x11000`, `0x11000 0x12000`, and `0x13000` respectively.

```
<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>
```

6.8.6 Banked address block

6.8.6.1 Schema

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



6.8.6.2 Description

The **addressBlock** element inside a bank element describes a single, contiguous block of memory that is part of a bank. The **addressBlock** element contains an **id** (optional) attribute that assigns a unique identifier to the containing element for reference throughout the containing description. **addressBlock** contains the following mandatory and optional elements.

a) ***nameGroup*** group is defined in C.1. The **name** of the **addressBlock**, **subspaceMap**, and **bank** shall be unique within the containing **bank** element.

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

1) **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** or **localMemoryMap/addressUnitBits** element. The type of this element is set to **scaledPositiveInteger**. The **range** element is configurable with attributes from ***long.prompt.att***, see C.12.

2) **width** (mandatory) is the bit width of a row in the address block. The type of this element is set to **nonNegativeInteger**. The **width** element is configurable with attributes from ***long.prompt.att***, see C.12.

c) ***memoryBlockData*** group contains information about usage, access, volatility, and other parameters. See 6.8.4.

d) ***registerData*** group contains information about the grouping of bits into registers and fields. See 6.10.2.

e) **vendorExtensions** (optional) adds any extra vendor-specific data related to the address block. See C.10.

NOTE—The ***bankedBlockType*** of an **addressBlock** element is almost identical to the ***addressBlockType*** of an **addressBlock** element (see 6.8.2); the only difference is there is no **baseAddress** and **typeIdentifier** in the ***bankedBlockType*** version.

See also: SCR 7.5.

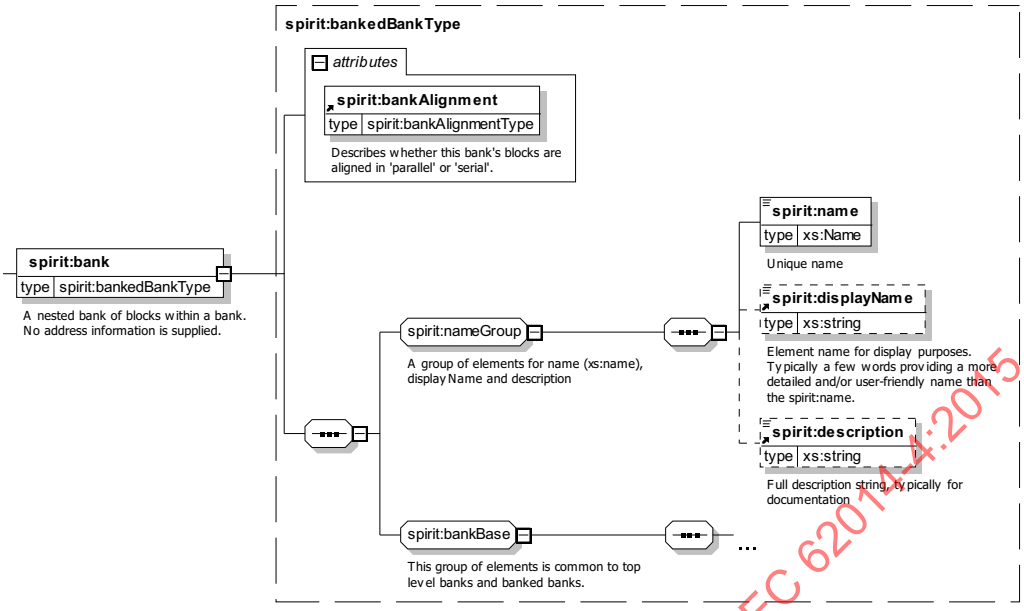
6.8.6.3 Example

See the example in 6.8.5.3.

6.8.7 Banked bank

6.8.7.1 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***.



6.8.7.2 Description

The **bank** element allows multiple address **blocks**, **banks**, or **subspaceMaps** to be to be concatenated together horizontally or vertically as 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 assigned an address in sequence.

b) ***nameGroup*** group is defined in C.1. The **name** of the **addressBlock**, **subspaceMap**, and **bank** shall be unique within the containing **bank** element.

c) The **bank** element of type ***bankedBankType*** contains the ***bankBase*** group. This group is defined inside the **bank** element of type ***addressBankType***. See 6.8.5. The effect of its inclusion here creates recursion, whereby banks maybe included inside banks included inside banks.

NOTE—A banked bank is similar to a bank in a memory map (see 6.8.5); the only difference is there is no **baseAddress** element in a **bank** of type ***bankedBankType***.

See also: SCR 8.2.

6.8.7.3 Example

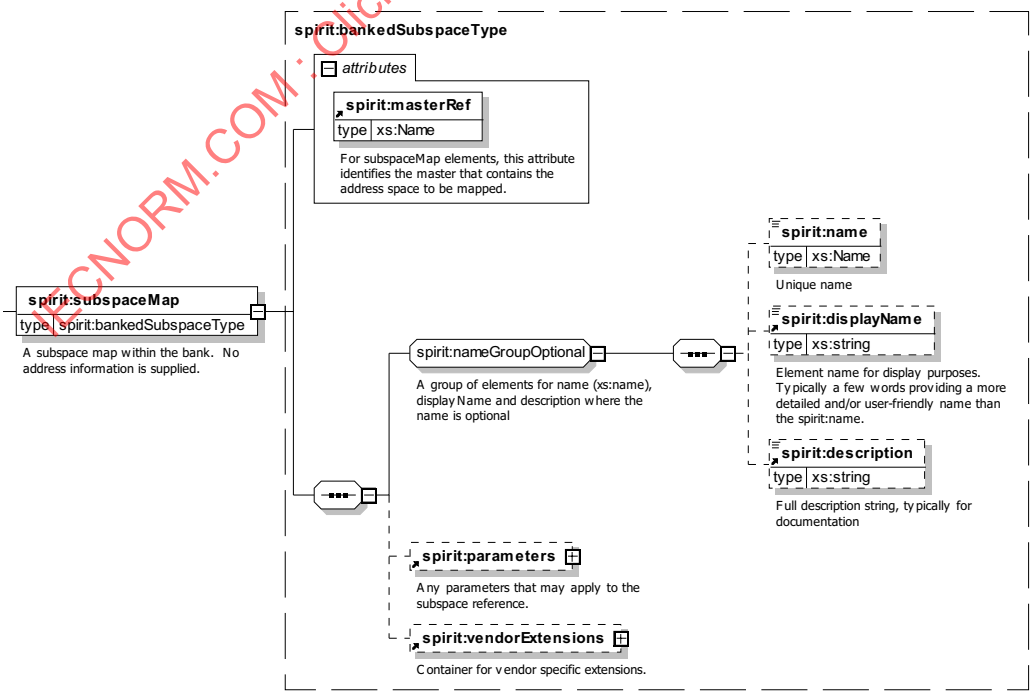
The following example shows a serial bank with two memory blocks of 1 k units of 32-bit data. The only address specified is `0x10000`, but this causes address block `ram0` and `bankRam1` to be mapped to addresses `0x10000` and `0x11000`, respectively. The memory bank `bankRam1` is made up of two parallel memory blocks each with 16 bits of data.

```
<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:bank bankAlignment="parallel">
            <spirit:name>bankRam1</spirit:name>
            <spirit:addressBlock>
                <spirit:name>ram1.0</spirit:name>
                <spirit:range>0x1000</spirit:range>
                <spirit:width>16</spirit:width>
            </spirit:addressBlock>
            <spirit:addressBlock>
                <spirit:name>ram1.1</spirit:name>
                <spirit:range>0x1000</spirit:range>
                <spirit:width>16</spirit:width>
            </spirit:addressBlock>
        </spirit:bank>
    </spirit:bank>
    <spirit:addressUnitBits>8</spirit:addressUnitBits>
</spirit:memoryMap>
```

6.8.8 Banked subspace

6.8.8.1 Schema

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



6.8.8.2 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 an interface mode of master (see 6.5.3). The master interface shall 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 is defined in C.2. The **name** of the **addressBlock**, **subspaceMap**, and **bank** shall be unique within the containing **bank** element.

c) **parameters** details any additional parameters that apply to the **subspaceMap**. See C.11.

d) **vendorExtensions** adds any extra vendor-specific data related to the **subspaceMap**. See C.10.

See also: SCR 8.2.

6.8.8.3 Example

The following example shows an address space from master `M1` mapped into the slave interface `S` memory map starting at address `0x0000`. An address space from master `M2` is mapped into the slave interface `S` memory map starting at address `0x1000`.

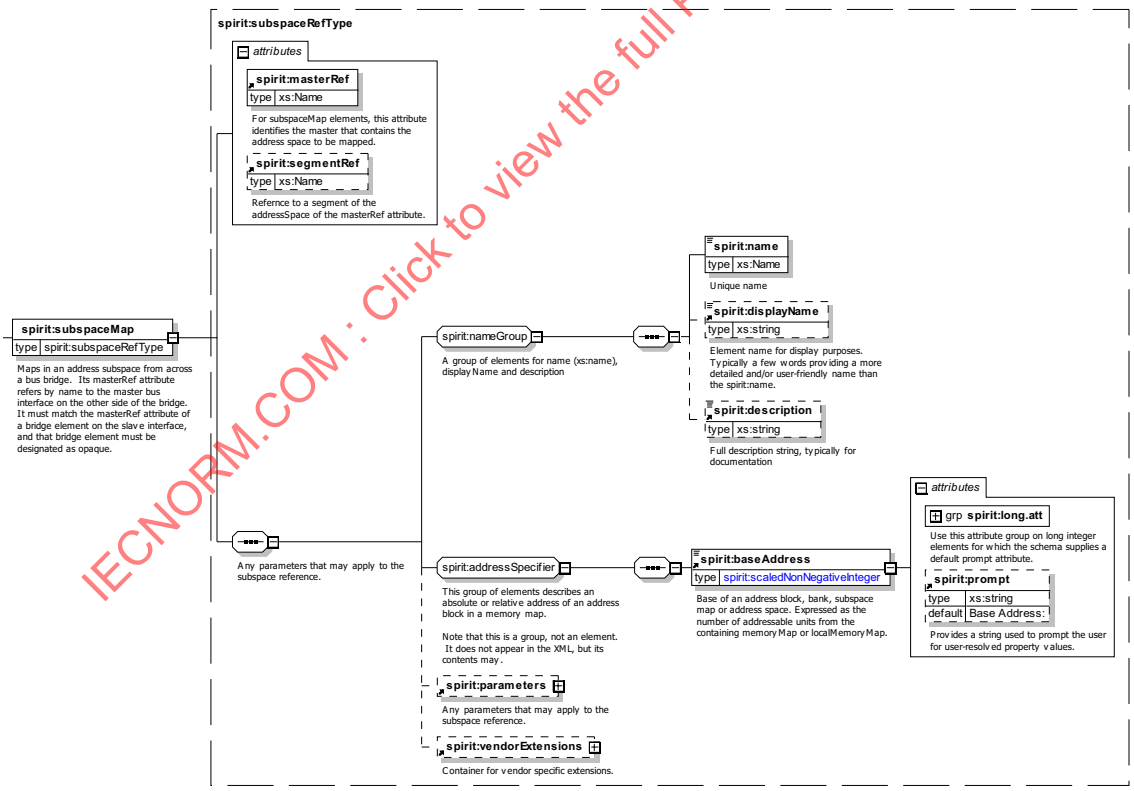
```
<spirit:component>…
    <spirit:busInterfaces>
        <spirit:busInterface>
            <spirit:name>M1</spirit:name>
            <spirit:master>
                <spirit:addressSpaceRef spirit:addressSpaceRef=”memAS1”\>
            </spirit:master>
        </spirit:busInterface>
        <spirit:busInterface>
            <spirit:name>M2</spirit:name>
            <spirit:master>
                <spirit:addressSpaceRef spirit:addressSpaceRef=”memAS2”\>
            </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:slave>
        </spirit:busInterface>
    </spirit:busInterfaces>
    <spirit:addressSpaces>
        <spirit:addressSpace>
            <spirit:name>memAS1</spirit:name>
            <spirit:range>0x1000</spirit:range>
            <spirit:width>32</spirit:width>
        </spirit:addressSpace>
        <spirit:addressSpace>
            <spirit:name>memAS2</spirit:name>
            <spirit:range>0x1000</spirit:range>
            <spirit:width>32</spirit:width>
        </spirit:addressSpace>
    </spirit:addressSpaces>
```

```
    <spirit:memoryMaps>
        <spirit:memoryMap>
            <spirit:name>memMap</spirit:name>
            <spirit:bank bankAlignment="serial">
                <spirit:name>memBank</spirit:name>
                <spirit:baseAddr baseAddress>0x0000</spirit:baseAddress>
                <spirit:subspaceMap spirit:masterRef="M1">
                    <spirit:name>submap1</spirit:name>
                </spirit:subspaceMap>
                <spirit:subspaceMap spirit:masterRef="M2">
                    <spirit:name>submap2</spirit:name>
                </spirit:subspaceMap>
            </spirit:bank>
        </spirit:memoryMap>
    </spirit:memoryMaps>
</spirit:component>
```

6.8.9 Subspace map

6.8.9.1 Schema

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



6.8.9.2 Description

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

a) **masterRef** (mandatory) attribute contains the name of the master interface whose address space needs to be mapped. This shall reference a bus interface name with an interface mode of master (see 6.5.3). The master interface shall also be referenced by a second interface through an **slave/bridge/masterRef** element, and the **bridge** element shall also have the **opaque** attribute set to **true**.

b) **segmentRef** (optional) references a **segment** in the **addressSpace** referred by the **masterRef** attribute. If the **segmentRef** attribute is not present, the entire **addressSpace** is presumed to be referenced.

c) ***nameGroup*** group is defined in C.1. The **name** of the **addressBlock**, **subspaceMap**, **bank,** and **memoryRemap** shall be unique within the containing **memoryMap**, **localMemoryMap**, or **memoryRemap** element.

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

baseAddress (mandatory) specifies the starting address of the block. The **baseAddress** is expressed in addressing units from the containing **memoryMap/addressUnitBits** or **localMemoryMap/addressUnitBits** element. The type of this element is set to ***scaledNonNegativeInteger***. The **baseAddress** element is configurable with attributes from ***long.att***, see C.12. The **prompt** attribute allows the setting of a string for the configuration and has a default value of **"Base Address:"**.

e) **parameters** (optional) details any additional parameters that apply to the **subspaceMap**. See C.11.

f) **vendorExtensions** (optional) adds any extra vendor-specific data related to the **subspaceMap**. See C.10.

See also: SCR 9.9 and SCR 3.18.

6.8.9.3 Example

The following example shows an address space from master `M1` mapped into the slave interface `S` memory map starting at address `0x0000`. An address space from master `M2` is mapped into the slave interface `S` memory map starting at address `0x1000`.

```
<spirit:component>…
   <spirit:busInterfaces>
      <spirit:busInterface>
         <spirit:name>M1</spirit:name>
         <spirit:master>
            <spirit:addressSpaceRef spirit:addressSpaceRef=”memAS1”\>
         </spirit:master>
      </spirit:busInterface>
      <spirit:busInterface>
         <spirit:name>M2</spirit:name>
         <spirit:master>
            <spirit:addressSpaceRef spirit:addressSpaceRef=”memAS2”\>
         </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:slave>
      </spirit:busInterface>
   </spirit:busInterfaces>
   <spirit:addressSpaces>
      <spirit:addressSpace>
         <spirit:name>memAS1</spirit:name>
         <spirit:range>0x1000</spirit:range>
```

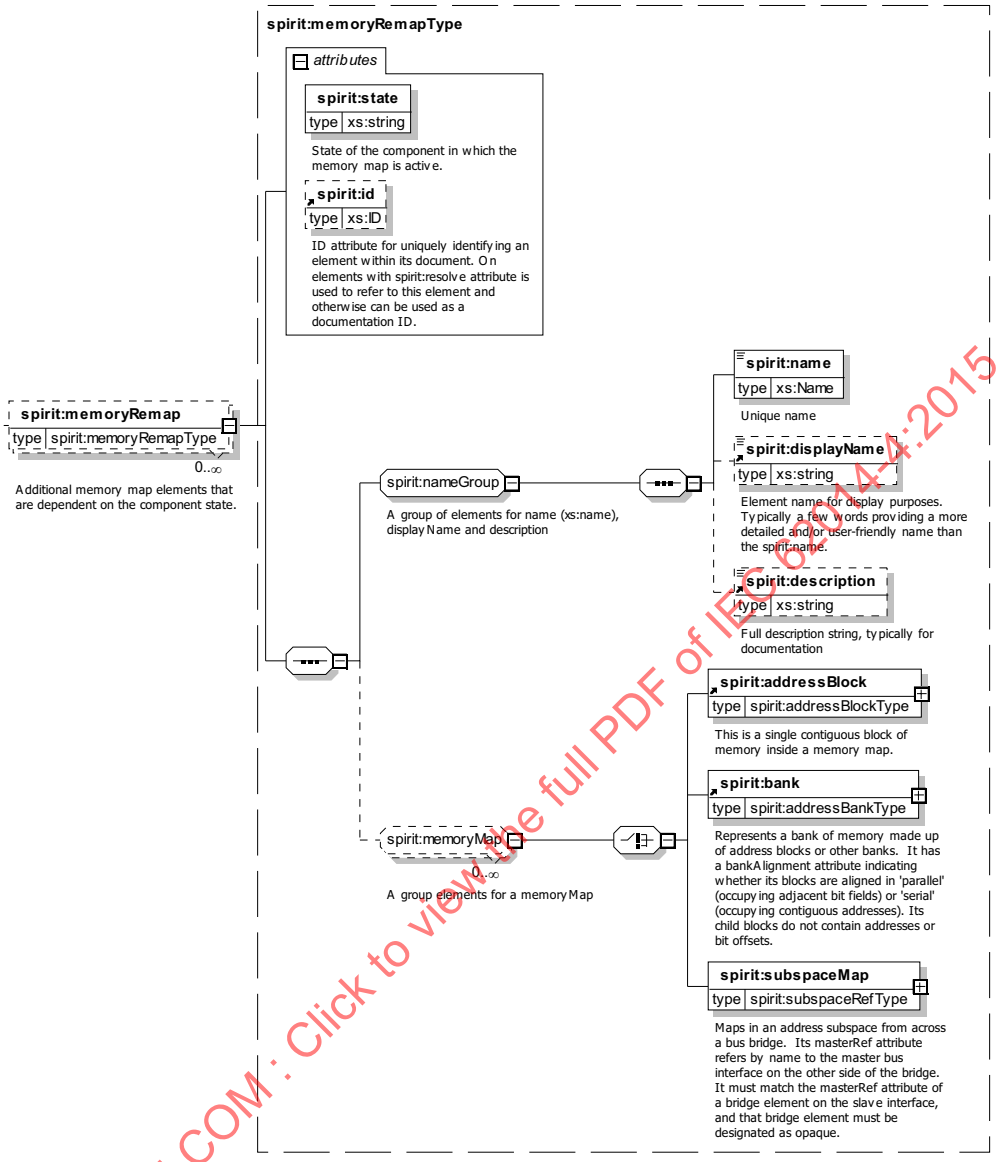
```
            <spirit:width>32</spirit:width>
         </spirit:addressSpace>
         <spirit:addressSpace>
            <spirit:name>memAS2</spirit:name>
            <spirit:range>0x1000</spirit:range>
            <spirit:width>32</spirit:width>
         </spirit:addressSpace>
      </spirit:addressSpaces>
      <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:memoryMap>
      </spirit:memoryMaps>
   </spirit:component>
```

6.9 Remapping

6.9.1 Memory remap

6.9.1.1 Schema

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



6.9.1.2 Description

The **memoryRemap** element describes additional **addressBlock**s, **bank**s, and **subspaceMap**s that are mapped on the referencing slave bus interface in a specific remap **state**. If multiple **memoryRemap/state** attributes are active, then the first **memoryRemap** listed shall be selected. The **memoryRemap** element contains an **id** (optional) attribute that assigns a unique identifier to the containing element for reference throughout the containing description. This element contains the following elements, attributes, and groups.

a) **state** attribute (mandatory) identifies the remap state name for which the optional memory map element are active. The **state** attribute shall reference a **remapState/name** in the containing description. The **state** attribute of all **memoryRemap** elements contained in a single **memoryMap** element shall be unique. The **state** attribute is of type ***string***. See 6.9.2.

b) ***nameGroup*** group is defined in C.1. The **name** of the **addressBlock**, **subspaceMap**, **bank**, and **memoryRemap** shall be unique within the containing **memoryMap** element.

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

1) **addressBlock** describes a single block. See 6.8.2.

2) **bank** represents a collections of address blocks, banks, or subspace maps. See 6.8.5.

3) **subspaceMap** maps the address subspaces of master interfaces into the slave's memory map. See 6.8.9.

6.9.1.3 Example

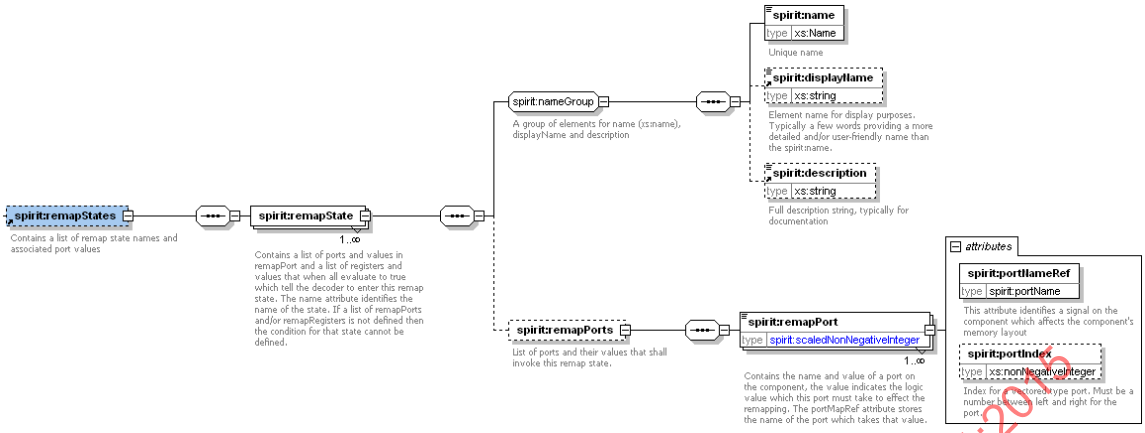
This is an example of a memory that is read-write in the normal state, but in state lock is remapped to be a read-only memory.

```
<spirit:memoryMaps>
   <spirit:memoryMap>
      <spirit:name>mmap1</spirit:name>
      <spirit:memoryReMap spirit:state="normal">
         <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:memoryReMap spirit: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>
```

6.9.2 Remap states

6.9.2.1 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.



6.9.2.2 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 subclause.

remapState contains the following elements and attributes.

a) ***nameGroup*** group is defined in C.1. The **name** element shall be unique within the containing **remapStates** element.

b) **remapPorts** (optional) contains a list of **remapPort** elements. **remapPort** (mandatory) specifies when the remap state gets effective. A collection of **remapPort** elements make up the condition for this remap state. All elements shall 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 following two attributes.

   1) **portNameRef** (mandatory) attribute is the name of the port in the containing description for which this logic value comparison is assigned. The **portNameRef** attribute is of type ***portName***. See 6.11.3.

   2) **portIndex** (optional) attribute references the index of a port in the containing description, when the port being referenced is vectored. The **portIndex** attribute is of type ***nonNegativeInteger***.

6.9.2.3 Example

This is an example of the **remapState** element with the state name of `boot`. The example specifies a remap state called `boot` is in effect when the port named `doRemap` gets the logic value of `0x01`, while another remap state called `normal` is in effect when the port gets the logic value of `0x00`.

```
<spirit:component>
   <spirit:remapStates>
      <spirit:remapState>
         <spirit:name>boot</spirit:name>
         <spirit:remapPorts>
            <spirit:remapPort spirit:portNameRef="doRemap">0x01
            </spirit:remapPort>
         </spirit:remapPorts>
      </spirit:remapState>
```

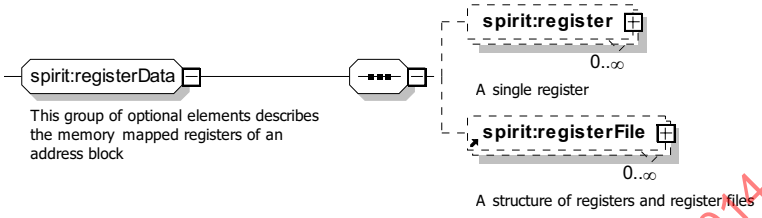
```
        <spirit:remapState>
            <spirit:name>normal</spirit:name>
            <spirit:remapPorts>
                <spirit:remapPort spirit:portNameRef="doRemap">0x00
                </spirit:remapPort>
            </spirit:remapPorts>
        </spirit:remapState>
    </spirit:remapStates >
</spirit:component>
```

6.10 Registers

6.10.1 Register data

6.10.1.1 Schema

The following schema details the information contained in the ***registerData*** group that may appear as an element inside the **addressBlock** element.



6.10.1.2 Description

The ***registerData*** group describes registers and register files. The containing **register/name** elements, the **register/alternateRegister/name** elements and the **registerFile/name** elements shall be unique within the containing **addressBlock** element. The ***registerData*** group contains these elements.

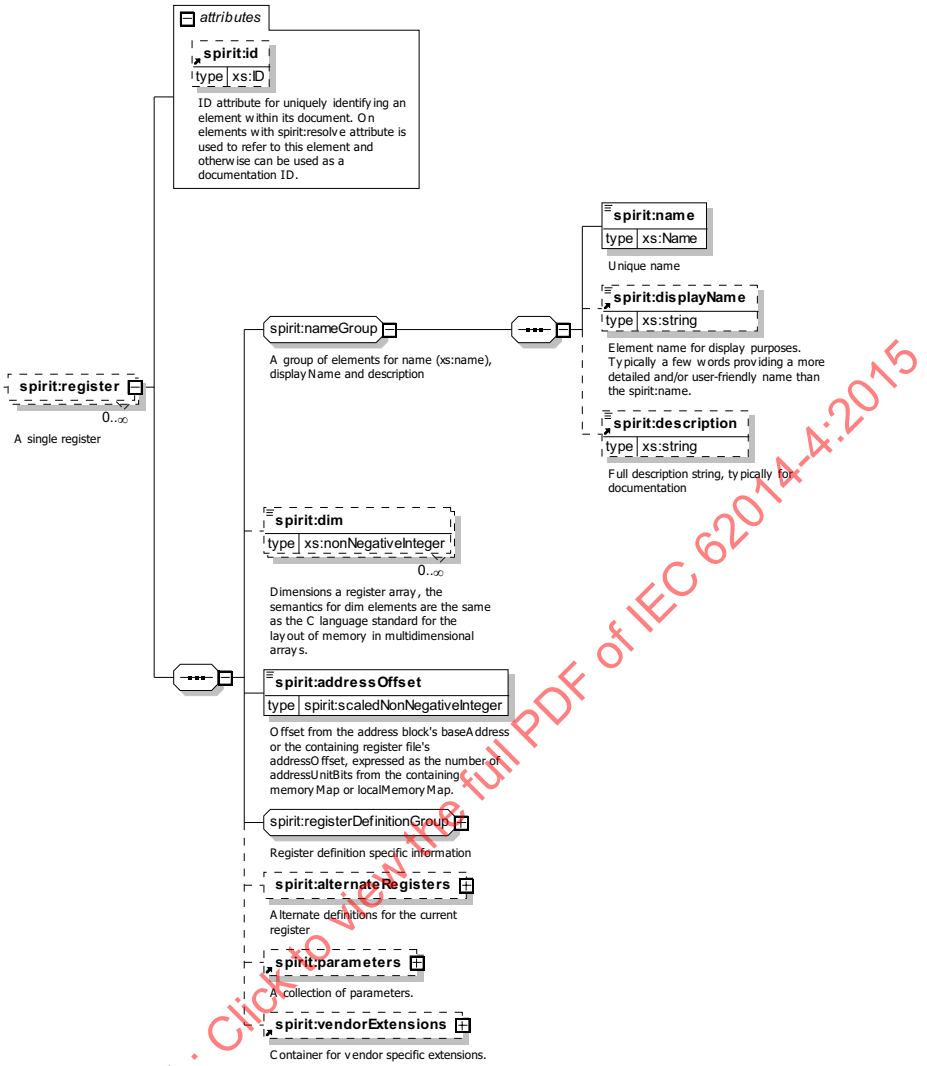
a) **register** (optional) defines a list of registers contained in this **addressBlock**. See 6.10.2.

b) **registerFile** (optional) defines a list of register files contained in this **addressBlock**. See 6.10.2.

6.10.2 Register

6.10.2.1 Schema

The following schema details the information contained in the **register** element, which is contained in the ***registerData*** group that may appear as an element inside the **addressBlock** element. This element describes a register.



6.10.2.2 Description

A **register** element describes a register in an address block or register file. The bits in the register are numbered from `size-1` down to `0`, with bit zero (`0`) being the least significant bit. The **register** element contains an **id** (optional) attribute that assigns a unique identifier to the containing element for reference throughout the containing description. **register** contains the following elements.

a) ***nameGroup*** group is defined in C.1. The **register/name**, **registerFile/name**, and **register/alternateRegisters/alternateRegister/name** element shall be unique within the containing **addressBlock** or **registerFile** element.

b) **dim** (optional) assigns an unbounded 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 IEEE Std 1666™-2005 [B4] (SystemC) language rules. The **dim** element is of type ***nonNegativeInteger***.

c) **addressOffset** (mandatory) describes the offset from the start of the containing **addressBlock** or **registerFile** element. The **addressOffset** is expressed in addressing units from the containing

memoryMap/addressUnitBits or **localMemoryMap/addressUnitBits** element. The **addressOffset** element is of type ***scaledNonNegativeInteger***.

d) ***registerDefinitionGroup*** group describes additional elements for a register. See 6.10.3

e) **alternateRegisters** (optional) describes alternate description for the containing register. See 6.10.4

f) **parameters** (optional) describes any parameter names and types when the register width can be parameterized. See C.11.

g) **vendorExtensions** (optional) adds any extra vendor-specific data related to this register. See C.10.

See also: SCR 7.1, SCR 7.2, SCR 7.3, SCR 7.4, SCR 7.5, SCR 7.7, SCR 7.8, SCR 7.9, SCR 7.13, SCR 8.3, SCR 8.4, SCR 8.5, SCR 8.7, SCR 8.8, and SCR 8.9.

6.10.2.3 Example

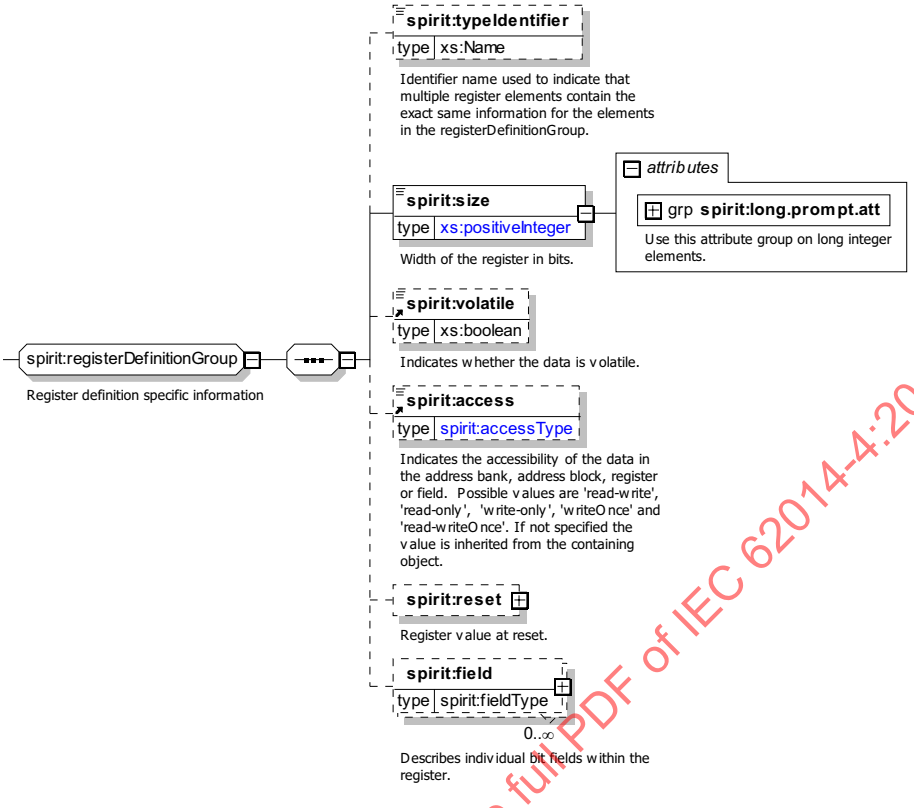
The following example shows a register with its subelements.

```
<spirit:register>
   <spirit:name>control</spirit:name>
   <spirit:description>Control register</spirit:description>
   <spirit:addressOffset>0x8</spirit:addressOffset>
   <spirit:size>32</spirit:size>
   <spirit:access>read-write</spirit:access>
   <spirit:field>
      <spirit:name>enable</spirit:name>
      <spirit:description>Enables the receiver</spirit:description>
      <spirit:bitOffset>0</spirit:bitOffset>
      <spirit:bitWidth>1</spirit:bitWidth>
   </spirit:field>
   <spirit:field>
   <!-- … -->
   </spirit:field>
</spirit:register>
```

6.10.3 Register definition group

6.10.3.1 Schema

The following schema details the information contained in the ***registerDefinitionGroup*** group, which is contained in the **register** element. This group describes register definition information.



6.10.3.2 Description

A ***registerDefinitionGroup*** group contains the following elements.

a) **typeIdentifier** (optional) indicates multiple register elements with the same **typeIdentifier** in the same description contain the exact same information for the elements in the **registerDefinitionsGroup**.

b) **size** (mandatory) is the width of the register, counting in bits. The type of this element is set to ***positiveInteger***. The **size** element is configurable with attributes from ***long.prompt.att***, see C.12.

c) **volatile** (optional) when **true** indicates in the case of a write followed by read, or in the case of two consecutive reads, there is no guarantee as to what is returned by the read on the second transaction or that this return value is consistent with the write or read of the first transaction. The element implies there is some additional mechanism by which this register can acquire new values other than by reads/writes/resets and other access methods known to IP-XACT. If this element is not present, no presumptions can be made about its value. The **volatile** element is of type ***boolean***.

d) **access** (optional) indicates the accessibility of the register. If this is not present, the **access** is inherited from the containing **addressBlock**. There are several choices.

1) **read-write**: Both read and write transactions may have an effect on this register. Write transactions may affect the contents of the register and read transactions return a value related to the values in the register.

2) **read-only**: A read transaction to this address returns a value related to the values in the register. A write transaction to this register has undefined results.

3) **write-only**: A write transaction to this address affects the contents of the register. A read transaction to this register has undefined results.

4) **read-writeOnce**: Both read and write transactions may have an effect on this register. Only the first write transaction, after an event that caused the reset value of the register to be loaded, may affect the contents of the register and read transactions return a value related to the values in the register.

5) **writeOnce**: Only the first write transaction, after an event that caused the reset value of the register to be loaded, affects the contents of the register. A read transaction to this register has undefined results.

e) **reset** (optional) indicates the value of the register's contents when the device is reset. See 6.10.7.

f) **field** (optional) describes any bit fields in a register. See 6.10.8.

See also: SCR 7.1, SCR 7.2, SCR 7.3, SCR 7.4, SCR 7.5, SCR 7.7, SCR 7.8, SCR 7.9, SCR 7.13, SCR 8.3, SCR 8.4, SCR 8.5, SCR 8.7, SCR 8.8, SCR 8.9, SCR 8.11, SCR 8.12, SCR 8.14, and SCR 8.15.

6.10.3.3 Example

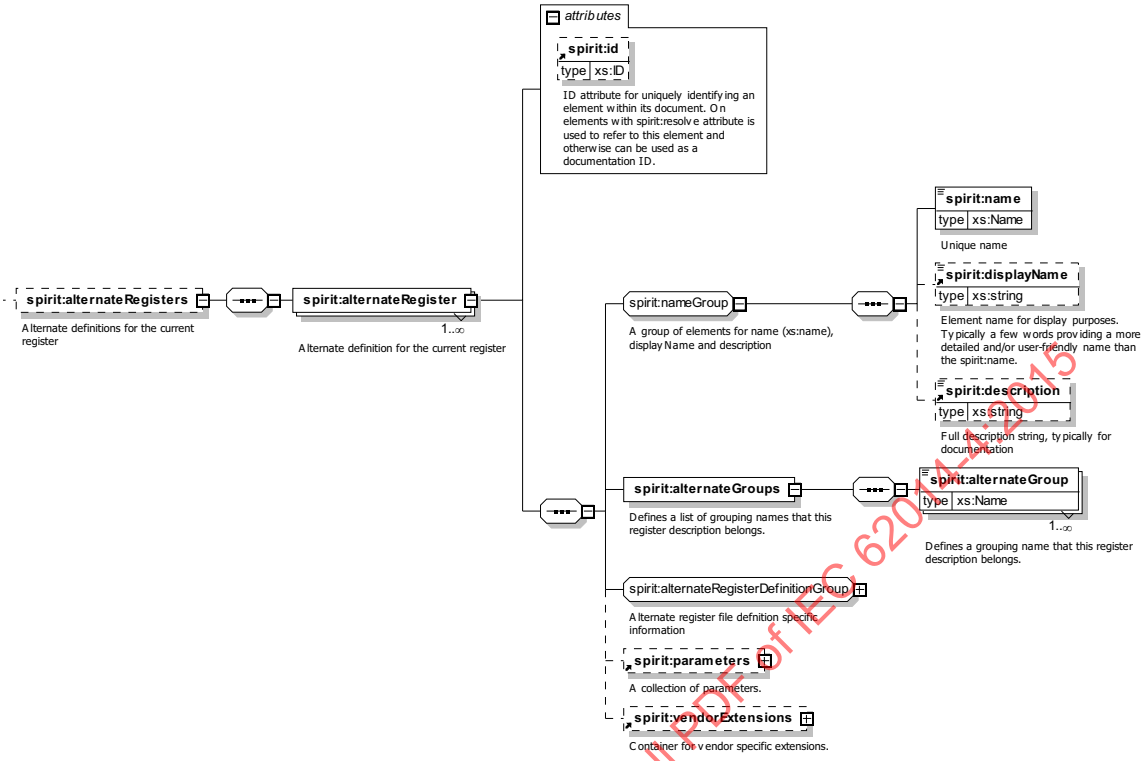
The following example shows a register with its subelements. The register contains a one bit field.

```
<spirit:register>
   <spirit:name>status</spirit:name>
   <spirit:description>Status register</spirit:description>
   <spirit:addressOffset>0x4</spirit:addressOffset>
   <spirit:size>32</spirit:size>
   <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:volatile>true</spirit:volatile>

   </spirit:field>
</spirit:register>
```

6.10.4 Alternate registers

6.10.4.1 Schema

The following schema details the information contained in the **alternateRegisters** element, which is contained in the **register** element that may appear as an element inside the **addressBlock** element. This element describes a list of alternate registers.



6.10.4.2 Description

The **alternateRegisters** (optional) element contains an unbounded list of **alternateRegister** elements. An **alternateRegister** element contains an alternate definition for the containing register. The **alternateRegister** element contains an **id** (optional) attribute that assigns a unique identifier to the containing element for reference throughout the containing description. **alternateRegister** contains the following elements.

a) ***nameGroup*** group is defined in C.1. The **register/name**, **registerFile/name**, and **register/alternateRegisters/alternateRegister/name** element shall be unique within the containing **addressBlock** or **registerFile** element.

b) **alternateGroups** (mandatory) defines an unbounded list of grouping names for which this alternate description belongs. **alternateGroup** (mandatory) defines a grouping name for this alternate register description. All **alternateGroup** elements shall be unique for each containing **register**. The **alternateGroup** element is of type ***Name***.

c) ***alternateRegisterDefinitionGroup*** group describes additional elements for an alternate register. See 6.10.3

d) **parameters** (optional) describes any parameter names and types when the register width can be parameterized. See C.11.

e) **vendorExtensions** (optional) adds any extra vendor-specific data related to this register. See C.10.

See also: SCR 7.1, SCR 7.2, SCR 7.3, SCR 7.4, SCR 7.7, SCR 7.8, SCR 7.9, SCR 7.13, SCR 8.3, SCR 8.4, SCR 8.5, SCR 8.7, SCR 8.8, SCR 8.9, SCR 8.11, SCR 8.12, SCR 8.14, and SCR 8.15.

6.10.4.3 Example

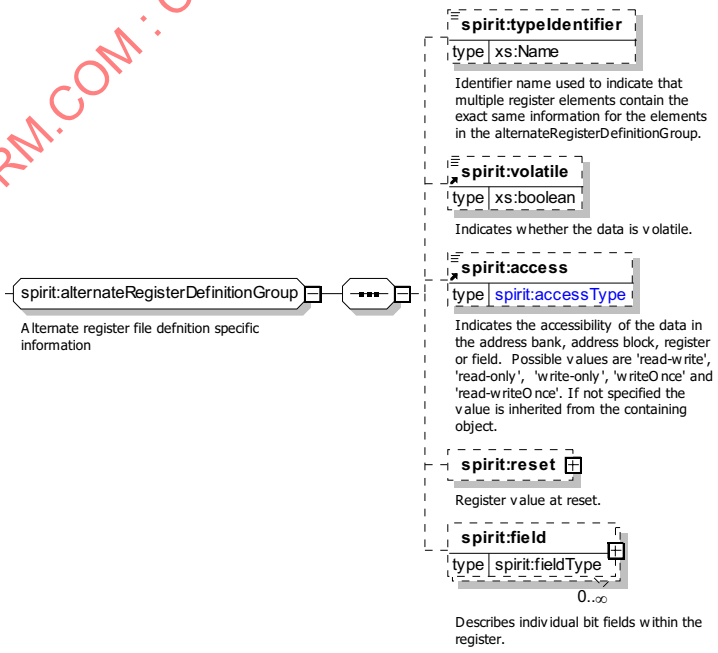
The following example shows a register with an alternate register definition that is a group called *transmit*.

```
<spirit:register>
   <spirit:name>control</spirit:name>
   <spirit:addressOffset>0x8</spirit:addressOffset>
   <spirit:size>32</spirit:size>
   <spirit:access>read-write</spirit:access>
   <spirit:field>
      <spirit:name>enable</spirit:name>
      <spirit:description>Enables the receiver</spirit:description>
      <spirit:bitOffset>0</spirit:bitOffset>
      <spirit:bitWidth>1</spirit:bitWidth>
   </spirit:field>
   <spirit:alternateRegisters>
      <spirit:alternateRegister>
         <spirit:name>control</spirit:name>
         <spirit:access>read-only</spirit:access>
         <spirit:field>
            <spirit:name>enable</spirit:name>
            <spirit:description>Enables the transmitter</
   spirit:description>
            <spirit:alternateGroups>
               <spirit:alternateGroup>transmit</spirit:alternateGroup>
            </spirit:alternateGroups>
         </spirit:field>
      </spirit:alternateRegister>
   </spirit:alternateRegisters>
</spirit:register>
```

6.10.5 Alternate register definition group

6.10.5.1 Schema

The following schema details the information contained in the ***alternateRegisterDefinitionGroup*** group, which is contained in the **alternateRegister** element. This group describes alternate register definition information.



6.10.5.2 Description

A ***alternateRegisterDefinitionGroup*** group contains the following elements.

a) **typeIdentifier** (optional) indicates multiple register elements with the same **typeIdentifier** in the same description contain the exact same information for the elements in the ***alternateRegisterDefinitionsGroup***.

b) **volatile** (optional) when **true** indicates in the case of a write followed by read, or in the case of two consecutive reads, there is no guarantee as to what is returned by the read on the second transaction or that this return value is consistent with the write or read of the first transaction. The element implies there is some additional mechanism by which this register can acquire new values other than by reads/writes/resets and other access methods known to IP-XACT. If this element is not present, no presumptions can be made about its value. The **volatile** element is of type ***boolean***.

c) **access** (optional) indicates the accessibility of the register. If this is not present, the **access** is inherited from the containing **addressBlock**. There are several choices.

   1) **read-write**: Both read and write transactions may have an effect on this register. Write transactions may affect the contents of the register and read transactions return a value related to the values in the register.

   2) **read-only**: A read transaction to this address returns a value related to the values in the register. A write transaction to this register has undefined results.

   3) **write-only**: A write transaction to this address affects the contents of the register. A read transaction to this register has undefined results.

   4) **read-writeOnce**: Both read and write transactions may have an effect on this register. Only the first write transaction, after power up, may affect the contents of the register and read transactions return a value related to the values in the register.

   5) **writeOnce**: Only the first write transaction, after power up, to this address affects the contents of the register. A read transaction to this register has undefined results.

d) **reset** (optional) indicates the value of the register's contents when the device is reset. See 6.10.7.

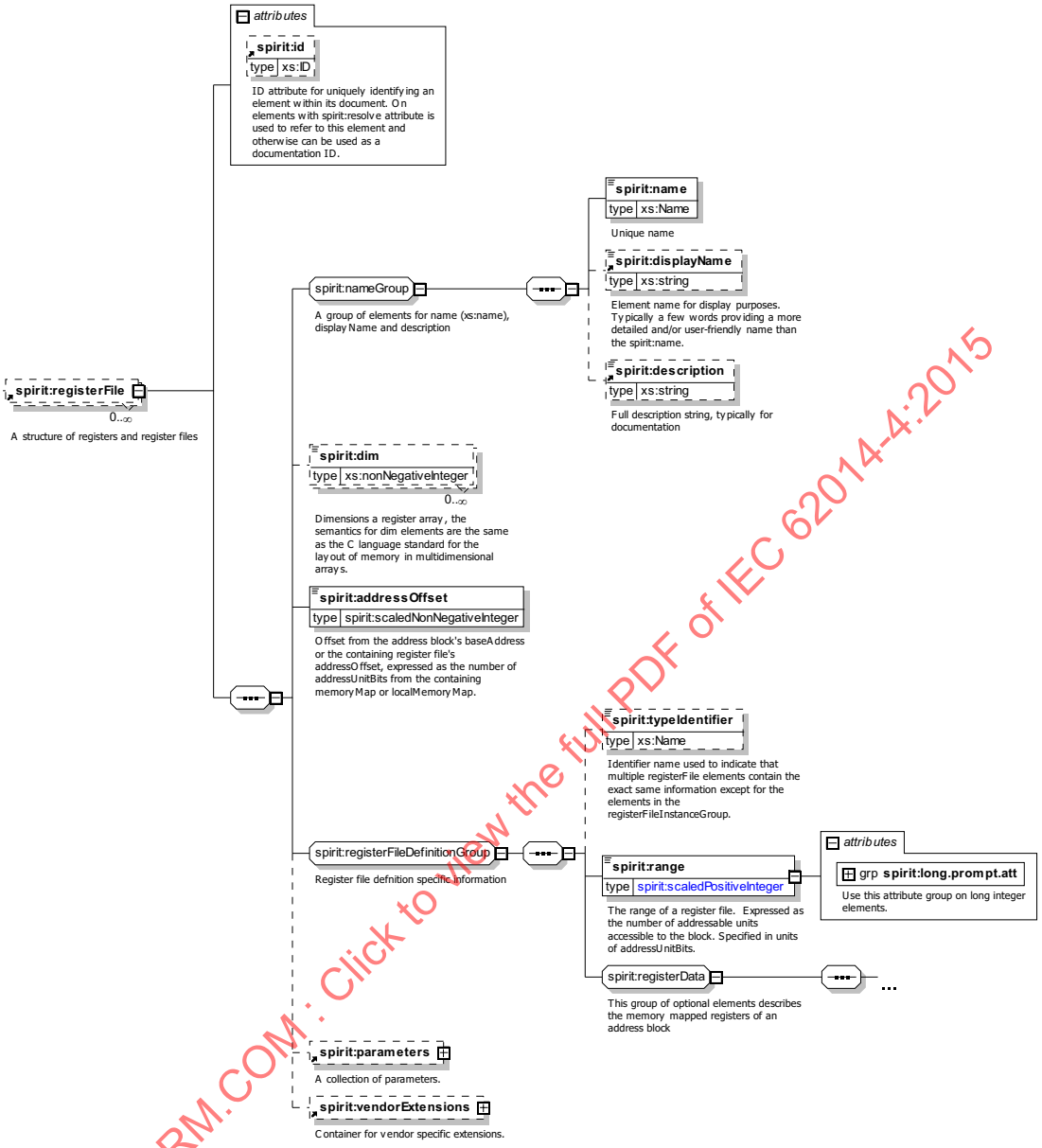
e) **field** (optional) describes any bit fields in a register. See 6.10.8.

See also: SCR 7.1, SCR 7.2, SCR 7.3, SCR 7.4, SCR 7.7, SCR 7.8, SCR 7.9, SCR 7.13, SCR 8.3, SCR 8.4, SCR 8.5, SCR 8.7, SCR 8.8, and SCR 8.9.

6.10.6 Register file

6.10.6.1 Schema

The following schema details the information contained in the **registerFile** element, which is contained in the ***registerData*** group that may appear as an element inside the **addressBlock** element. This element describes a register file.



6.10.6.2 Description

A **registerFile** element describes a grouping of registers in an address block or register file. The **registerFile** element contains an **id** (optional) attribute that assigns a unique identifier to the containing element for reference throughout the containing description. **registerFile** contains the following elements.

a) ***nameGroup*** group is defined in C.1. The **register/name**, **registerFile/name**, and **register/alternateRegisters/alternateRegister/name** element shall be unique within the containing **addressBlock** or **registerFile** element.

b) **dim** (optional) assigns an unbounded 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 IEEE Std 1666-2005 [B4] (SystemC) language rules. The **dim** element is of type ***nonNegativeInteger***.

c) **addressOffset** (mandatory) describes the offset from the start of the containing **addressBlock** or **registerFile** element. The **addressOffset** is expressed in addressing units from the containing **memoryMap/addressUnitBits** or **localMemoryMap/addressUnitBits** element. The **addressOffset** element is of type ***scaledNonNegativeInteger***.

d) ***registerFileDefinitionGroup*** group describes additional elements for a register file.

1) **typeIdentifier** (optional) indicates multiple register elements with the same **typeIdentifier** in the same description contain the exact same information for the elements in the **registerDefinitionsGroup**.

2) **range** (mandatory) gives the range of a register file. This is expressed as the number of addressable units of the register file. 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 C.12.

3) ***registerData*** group contains information about the grouping of bits into registers and fields. See 6.10.2.

e) **parameters** (optional) describes any parameter names and types when the register width can be parameterized. See C.11.

f) **vendorExtensions** (optional) adds any extra vendor-specific data related to this register. See C.10.

See also: SCR 7.6, SCR 7.7, and SCR 7.14.

6.10.6.3 Example

The following example shows a register file within an address block starting at address `0x200`. The register file is 32 bytes in length and contains two registers at an absolute address of `0x200` and `0x204` within the address block.

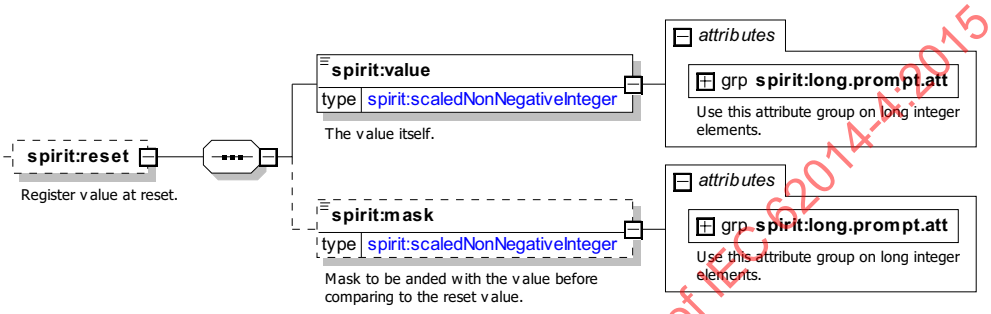
```
<spirit:addressBlock>
   <spirit:name>abname</spirit:name>
   <spirit:baseAddress>0x0</spirit:baseAddress>
   <spirit:range>0x1000</spirit:range>
   <spirit:width>32</spirit:width>
   <spirit:registerFile>
      <spirit:name>status</spirit:name>
      <spirit:description>Status register</spirit:description>
      <spirit:addressOffset>0x200</spirit:addressOffset>
      <spirit:range>32</spirit:range>
      <spirit:register>
         <spirit:name>control</spirit:name>
         <spirit:addressOffset>0x0</spirit:addressOffset>
         <spirit:size>32</spirit:size>
         <spirit:access>read-write</spirit:access>
         <spirit:field>
         <!-- ... -->
         </spirit:field>
      </spirit:register>
      <spirit:register>
         <spirit:name>status</spirit:name>
         <spirit:addressOffset>0x4</spirit:addressOffset>
         <spirit:size>32</spirit:size>
         <spirit:access>read-only</spirit:access>
         <spirit:field>
         <!-- ... -->
         </spirit:field>
      </spirit:register>
```

```
    </spirit:registerFile>
    <spirit:addressUnitBits>8</spirit:addressUnitBits>
</spirit:addressBlock>
```

6.10.7 Register reset value

6.10.7.1 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.



6.10.7.2 Description

The **reset** element describes the value of a register at reset. It has the following 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 C.12.

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 C.12.

   A `1` bit in the **mask** means the corresponding bit of the register has a known reset value; a `0` bit means it does not. All bits of the **value** that 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.

6.10.7.3 Example

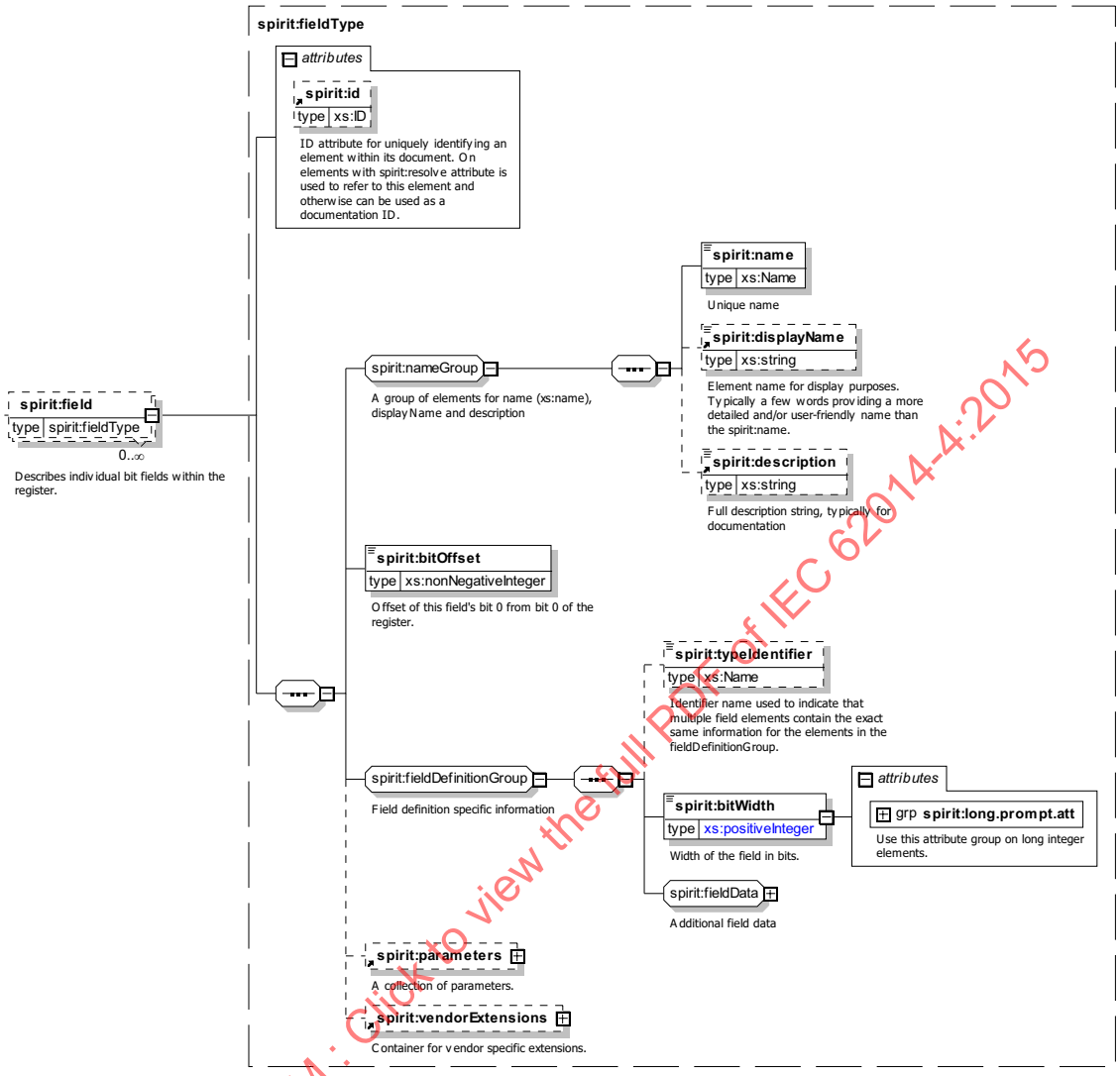
The following example shows a reset value. Any register with this reset value has bit `7` and bits `5` down to `1` set to logic `0`, and bits `6` and `0` set to a logic `1` on reset.

```
<spirit:reset>
    <spirit:value>0x41</spirit:value>
    <spirit:mask>0xFF</spirit:mask>
</spirit:reset>
```

6.10.8 Register bit fields

6.10.8.1 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.



6.10.8.2 Description

A **field** element of a **register** describes a smaller bit field of a register. The **field** element contains an **id** (optional) attribute that assigns a unique identifier to the containing element for reference throughout the containing description. **field** contains the following elements.

a) ***nameGroup*** group is defined in C.1. The **name** element shall be unique within the containing **register** or **alternateRegister** element.

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) ***fieldDefinitionGroup*** group describes additional elements for a field.

1) **typeIdentifier** (optional) indicates multiple fields elements with the same **typeIdentifier** in the same description contain the exact same information for the elements in ***fieldDefinitionsGroup***.

2) **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 C.12.

3) ***fieldData*** group describes additional elements for a field. See 6.10.10.

d) **parameters** (optional) details any additional parameters that describe the **field** for generator usage. See C.11.

e) **vendorExtensions** (optional) adds any extra vendor-specific data related to this field. See C.10.

See also: SCR 7.2, SCR 7.4, SCR 7.9, SCR 7.10, SCR 7.11, and SCR 7.12.

6.10.8.3 Example

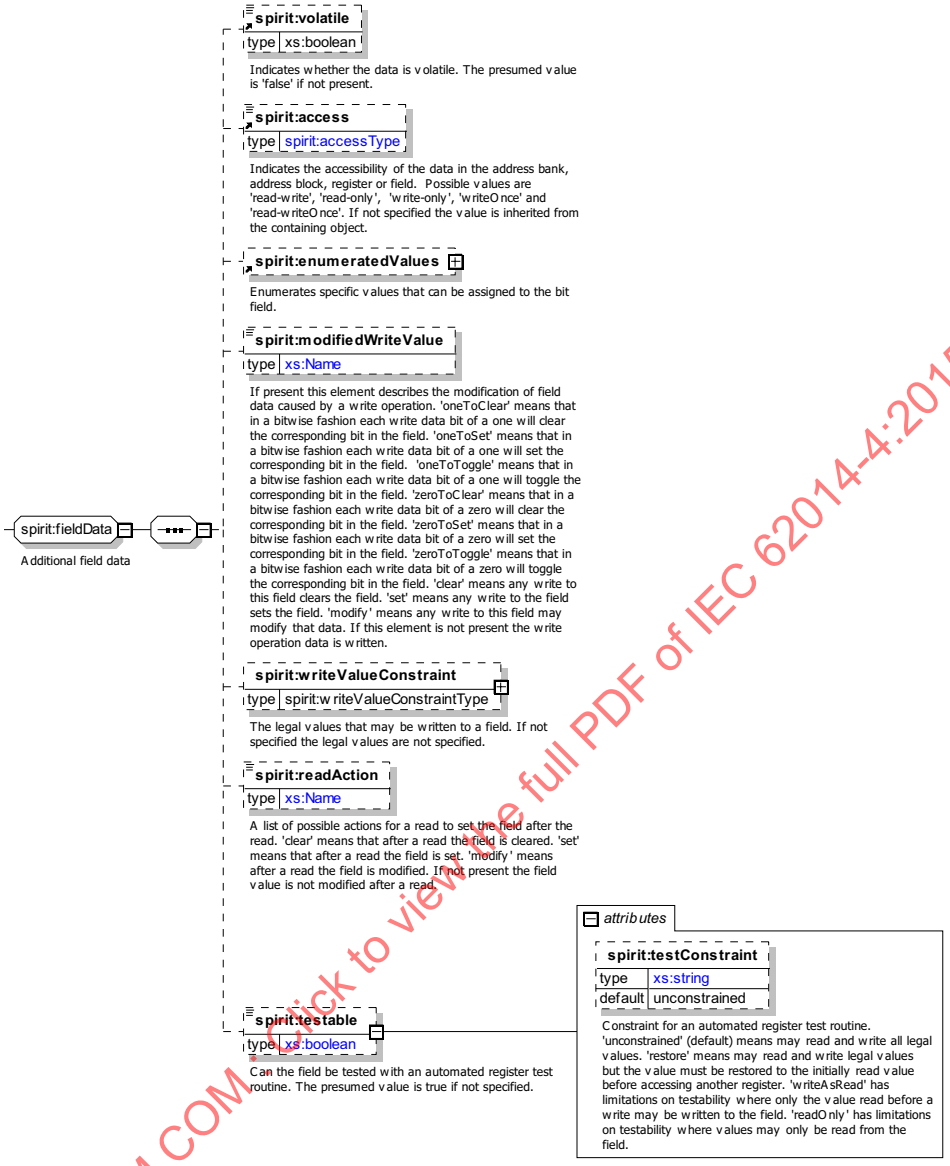
The following example shows a 1-bit field with its subelements.

```
<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>2</spirit:bitOffset>
   <spirit:bitWidth>1</spirit:bitWidth>
</spirit:field>
```

6.10.9 Field data group

6.10.9.1 Schema

The following schema details the information contained in the ***fieldData*** group, which is contained inside the **field** element. This group describes the optional properties of a **field**.



6.10.9.2 Description

The ***fieldData*** group contains the following elements.

a) **volatile** (optional) when **true** indicates in the case of a write followed by read, or in the case of two consecutive reads, there is no guarantee as to what is returned by the read on the second transaction or that this return value is consistent with the write or read of the first transaction. The element implies there is some additional mechanism by which this field can acquire new values other than by reads/writes/resets and other access methods known to IP-XACT. If this element is not present, it is presumed to be **false**. The **volatile** element is of type ***boolean***.

b) **access** (optional) indicates the accessibility of the field. If this is not present, the **access** is inherited from the containing **register**. There are several choices.

1) **read-write**: Both read and write transactions may have an effect on this field. Write transactions may affect the contents of the field and read transactions return a value related to the values in the field.

2) **read-only**: A read transaction to this address returns a value related to the values in the field. A write transaction to this field has undefined results.

3) **write-only**: A write transaction to this address affects the contents of the field. A read transaction to this field has undefined results.

4) **read-writeOnce**: Both read and write transactions may have an effect on this field. Only the first write transaction, after power up, may affect the contents of the field and read transactions return a value related to the values in the field.

5) **writeOnce**: Only the first write transaction, after power up, to this address affects the contents of the field. A read transaction to this field has undefined results.

c) **enumeratedValues** (optional) describes a name for different values of a field. See 6.10.10.

d) **modifiedWriteValue** (optional) element to describe the manipulation of data written to a field. The value shall be one of **oneToClear**, **oneToSet**, **oneToToggle**, **zeroToClear**, **zeroToSet**, **zeroToToggle**, **clear**, **set**, or **modify**. If the **modifiedWriteValue** element is not specified, the value written to the field is the value stored in the field.

oneToClear means in a bitwise fashion each write data bit of a one shall clear (set to zero) the corresponding bit in the field.

oneToSet means in a bitwise fashion each write data bit of a one shall set (set to one) the corresponding bit in the field.

oneToToggle means in a bitwise fashion each write data bit of a one shall toggle the corresponding bit in the field.

zeroToClear means in a bitwise fashion each write data bit of a zero shall clear (set to zero) the corresponding bit in the field.

zeroToSet means in a bitwise fashion each write data bit of a zero shall set (set to one) the corresponding bit in the field.

zeroToToggle means in a bitwise fashion each write data bit of a zero shall toggle the corresponding bit in the field.

clear means after a write operation all bits in the field are cleared (set to zero).

set means that after a write operation all bits in the field are set (set to one).

modify means that after a write operation all bits in the field may be modified.

e) **writeValueConstraint** (optional) describes a set of constraint values that are the only values that can be written to this field. If **writeValueConstraint** is not present, no constraint values are defined for this field. See 6.10.11.

f) **readAction** (optional) describes an action that happens to a field after a read operation happens. **clear** indicates the field is cleared after a read operation. **set** indicates the field is set after a read operation. **modify** indicates the field is modified in some way after a read operation. If **readAction** not specified, the field is not modified after a read operation.

g) **testable** (optional) defines if the field is testable by a simple automated register test. If this is not present, **testable** is presumed to be **true**. The **testable** element is of type ***boolean***.

testConstraint (optional) attribute defines the constraint for the field during a simple automated register test.

unConstrained (default) indicates there are no restrictions on the data that may be written or read from the field. **restore** indicates the field's value shall be restored to the original value before accessing another register. **writeAsRead** indicates the field shall only be written to a value just previously read from the field. **readOnly** indicates the field shall only be read.

6.10.9.3 Example

The following example describes a field with a write behavior that sets a field bit if the value written is a `1` and a read behavior that reads the written value and afterwards clears the register.

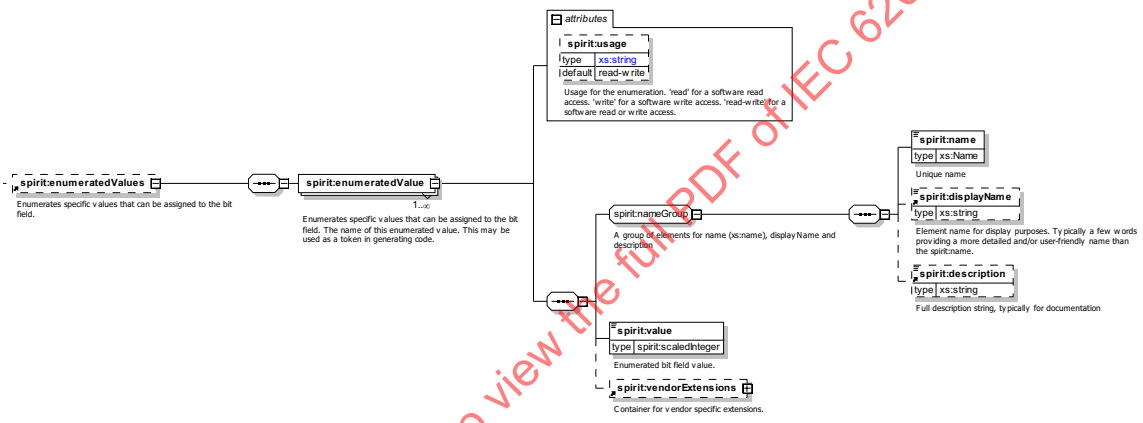
```
<spirit:field>
    <spirit:name>interrupt</spirit:name>
    <spirit:bitOffset>0</spirit:bitOffset>
    <spirit:bitWidth>1</spirit:bitWidth>
    <spirit:modifiedWriteValue>oneToSet</spirit:modifiedWriteValue>
    <spirit:readAction>clear</spirit:readAction>
</spirit:field>
```

See also: SCR 7.2, SCR 7.4, SCR 7.9, SCR 7.10, SCR 7.11, and SCR 7.12.

6.10.10 Enumeration values

6.10.10.1 Schema

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



6.10.10.2 Description

The **enumeratedValues** element describes a list of name and values pairs for the given field.

a) **usage** (optional) attribute defines the software access condition under which this name value pair is valid. Possible values are **read**, **write**, and **read-write** (default).

b) ***nameGroup*** group is defined in C.1. All **name** elements shall be unique within the containing **enumeratedValues** element.

c) **value** (mandatory) defines the value to assign to the specified name. The **value** element is of type ***scaledInteger***.

d) **vendorExtensions** (optional) adds any extra vendor-specific data related to this enumeration. See C.10.

6.10.10.3 Example

The following example shows two enumerated values for a 1-bit field: `0` for `oddParity` and `1` for `evenParity`.

```
<spirit:enumeratedValues>
   <spirit:enumeratedValue>
      <spirit:name>oddParity</spirit:name>
      <spirit:description>oddParity</spirit:description>
```

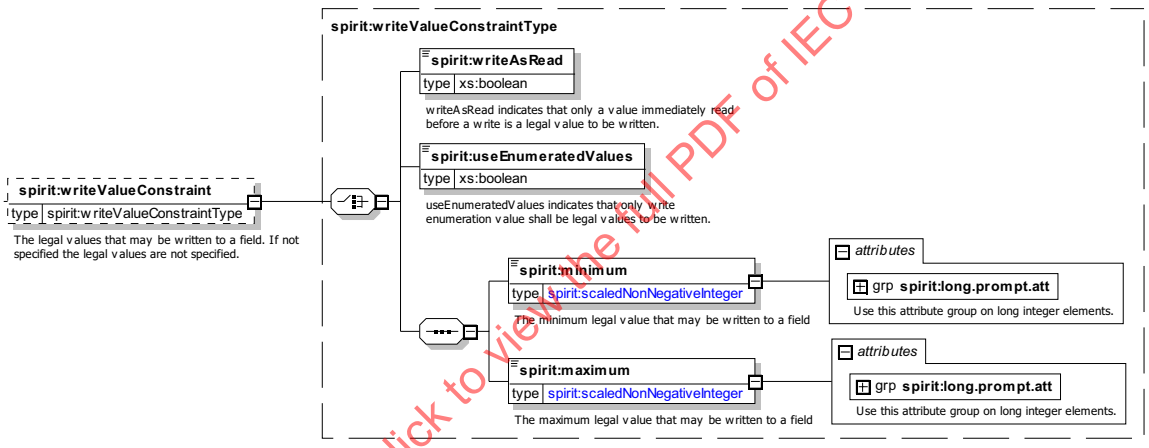
```
            <spirit:value>0</spirit:value>
        </spirit:enumeratedValue>
        <spirit:enumeratedValue>
            <spirit:name>evenParity</spirit:name>
            <spirit:description>evenParity</spirit:description>
            <spirit:value>1</spirit:value>
        </spirit:enumeratedValue>
    </spirit:enumeratedValues>
```

See also: SCR 7.10 and SCR 7.11.

6.10.11 Write value constraint

6.10.11.1 Schema

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



6.10.11.2 Description

The **writeValueConstraint** element describes a set of constraint values that are the only values that can be written to this field. If **writeValueConstraint** is not present, the legal values for this field are not defined.

a) **writeAsRead** (mandatory) if **true** implies the only legal value to write to this field is a value previously read from this field. If **writeAsRead** is not present, it is presumed to be **false**. The **writeAsRead** element is of type ***boolean***.

b) **useEnumeratedValues** (mandatory) if **true** implies the only legal values to write to this field are the values specified in the **useEnumeratedValues** element for this field. If **useEnumeratedValues** is not present, it is presumed to be **false**. The **useEnumeratedValues** element is of type ***boolean***.

c) **minimum** (mandatory) contains the minimum value that may be written to this field. The **minimum** element is of type ***scaledNonNegativeInteger***. The **minimum** element is configurable with attributes from ***long.prompt.att***, see C.12.

d) **maximum** (mandatory) contains the maximum value that may be written to this field. The **maximum** element is of type ***scaledNonNegativeInteger***. The **maximum** element is configurable with attributes from ***long.prompt.att***, see C.12.

See also: SCR 7.10 and SCR 7.11.

6.10.11.3 Example

The following example is for a two-bit field that only allows the values `0`, `1`, and `2` to be written.

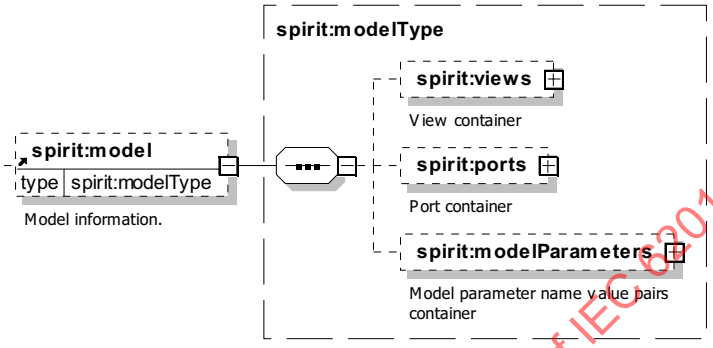
```
<spirit:writeValueConstraint>
       <spirit:minimum>0x0</spirit:minimum>
       <spirit:maximum>0x2</spirit:maximum>
<spirit:writeValueConstraint>
```

6.11 Models

6.11.1 Model

6.11.1.1 Schema

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



6.11.1.2 Description

The **model** element describes the views, ports, and model-related parameters of a component. 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 software 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 6.11.2.

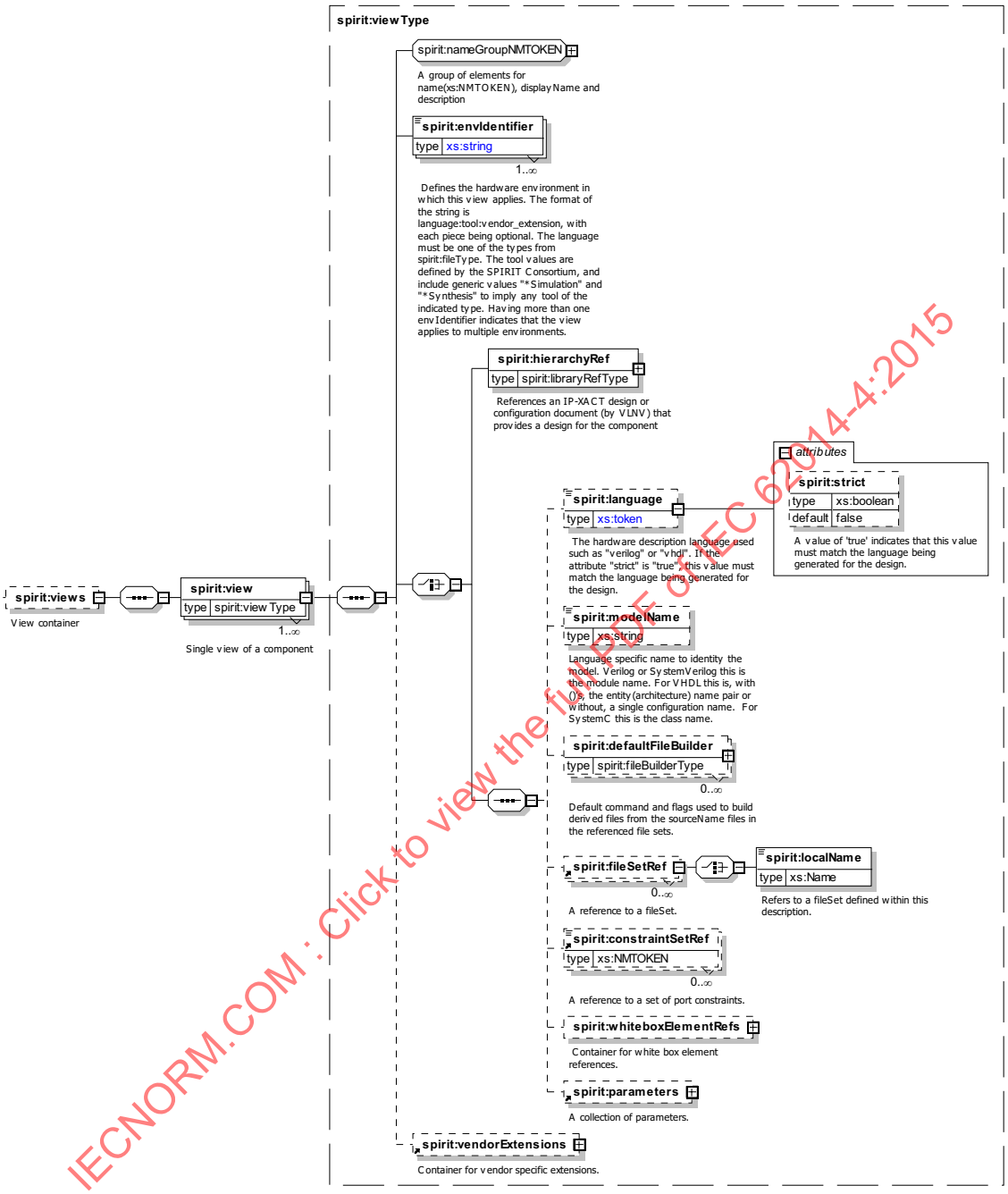
b) **ports** (optional) contains the list of ports for this object. A ports is an external connection from the object. An object may only have one set of ports that shall be valid for all views. See 6.11.3.

c) **modelParameters** (optional) contains a list of parameters that are needed to configure a model implementation specified in a view. An object shall only have one set of model parameters that are valid for all views. See 6.11.20.

6.11.2 Views

6.11.2.1 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.



6.11.2.2 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 is detailed in C.4. The **name** elements shall be unique within the containing **views** element.

b) **envIdentifier** (mandatory) 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 colons [**:**] in the format of *Language***:***Tool***:***VendorSpecific*. The regular expression that

is used to check the string is [A-Za-z0-9_+*\.]*:[A-Za-z0-9_+*\.]*:[A-Za-z0-9_+*\.]* The sections are:

1) *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.

2) *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, including `*Simulation` and `*Synthesis`. To support transportability of created data files, it is important to use the published, generally recognized, tool designation when referencing a tool. See IP-XACT standard tool names for **envIdentifier** [B14].

3) *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) All of the information for a **view** shall be in the containing **component**. Specifically, the **fileSets** that are referenced in a **view** shall contain entries for all of the required files. If a **view** in the **component** contains a **hierarchyRef**, other **view**s shall not assume the inclusion of files in a **fileSet** referenced through that **hierarchyRef**. The implementation details for this **view** has two possibilities.

   The first possibility is a hierarchical view that uses the **hierarchyRef** element. In this case:

   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***, it contains four attributes to specify a unique VLNV. See C.7.

d) The second possibility is a non-hierarchical view that references a file set. In this case:

   1) **language** (optional) specifies the HDL 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. If this attribute is not present, its effective value is **false**.

   2) **modelName** (optional) is a language-specific identifier of the model. For Verilog or SystemVerilog, this is the module name. For VHDL, this is, with `()`'s, the entity (architecture) name pair or, without `()`'s, a configuration name. For SystemC, this is the `sc_module` class name. 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**. This contains all the same elements as the element **fileBuilder**. See 6.13.5.

4) **fileSetRef** (optional) is an unbounded list of references to **fileSet name**s within the containing document or another document referenced by the **VLNV**. See C.8.

5) **constraintSetRef** (optional) is an unbounded list of references to constraint sets, valid timing constraints for a view. **constraintsSets** are only defined for **wire** style **ports**. The **constraintSetRef** element is of type ***NMTOKEN***. See 6.11.9.

6) **whiteboxElementRefs** (optional) contains references to white box elements of a component that are valid for this view. If the view contains an implementation of any of the white box 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 6.15.

7) **parameters** (optional) details any additional parameters that describe the **view**. See C.11.

e) **vendorExtensions** (optional) adds any extra vendor-specific data related to the view. See C.10.

See also: SCR 14.3.

6.11.2.3 Example

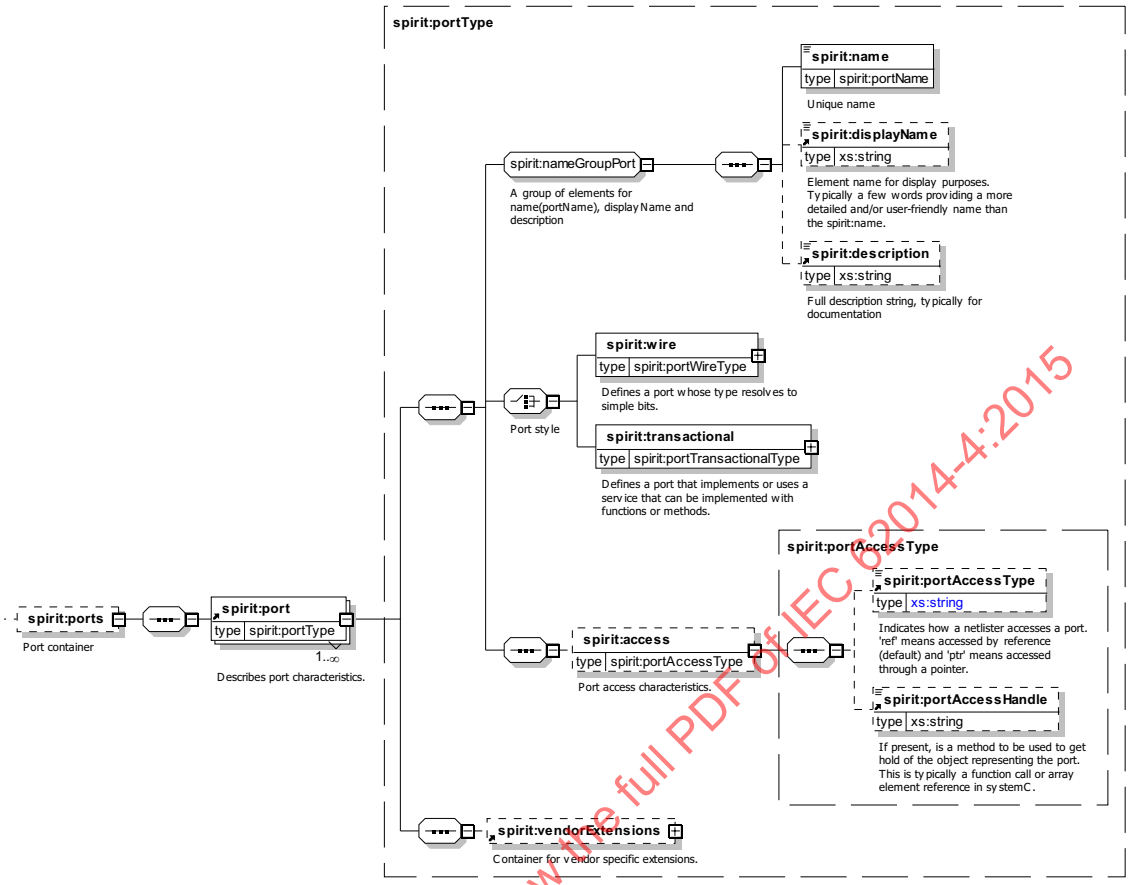
The following is an example of a non-hierarchical **view** element with the name of `vhdlsource`.

```
<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>
         <spirit:localName>fs-vhdlSource</spirit:localName>
      </spirit:fileSetRef>
      <spirit:constraintSetRef>normal</spirit:constraintSetRef>
   </spirit:view>
</spirit:views>
```

6.11.3 Component ports

6.11.3.1 Schema

The following schema defines the information contained in the **ports** element, which may appear within a **component**.



6.11.3.2 Description

The **ports** element defines an unbounded list of **port** elements. Each **port** element describes a single external port on the component.

a) ***nameGroupPort*** group is defined in C.4. The **name** elements shall be unique within the containing **ports** element.

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 6.11.4.2.

   2) **transactional** (mandatory) defines all other styles of ports, typically used for TLM. See 6.11.16.

c) **access** (optional) defines the access for a port.

   1) **portAccessType** (optional) indicates to a netlister how to access the port. The **portAccessType** has one of two possible values **ref** or **ptr**. If **ref** (the default), a netlister should access the port directly and, if **ptr**, it should access the port with a pointer.

   2) **portAccessHandle** (optional) indicates to a netlister the method to be used to access the object representing the port. This is typically a function call or array element reference in IEEE Std 1666-2005 [B4] (SystemC). The **portAccessHandle** is of type ***string***.

d) **vendorExtensions** (optional) adds any extra vendor-specific data related to the port. See C.10.

6.11.3.3 Example

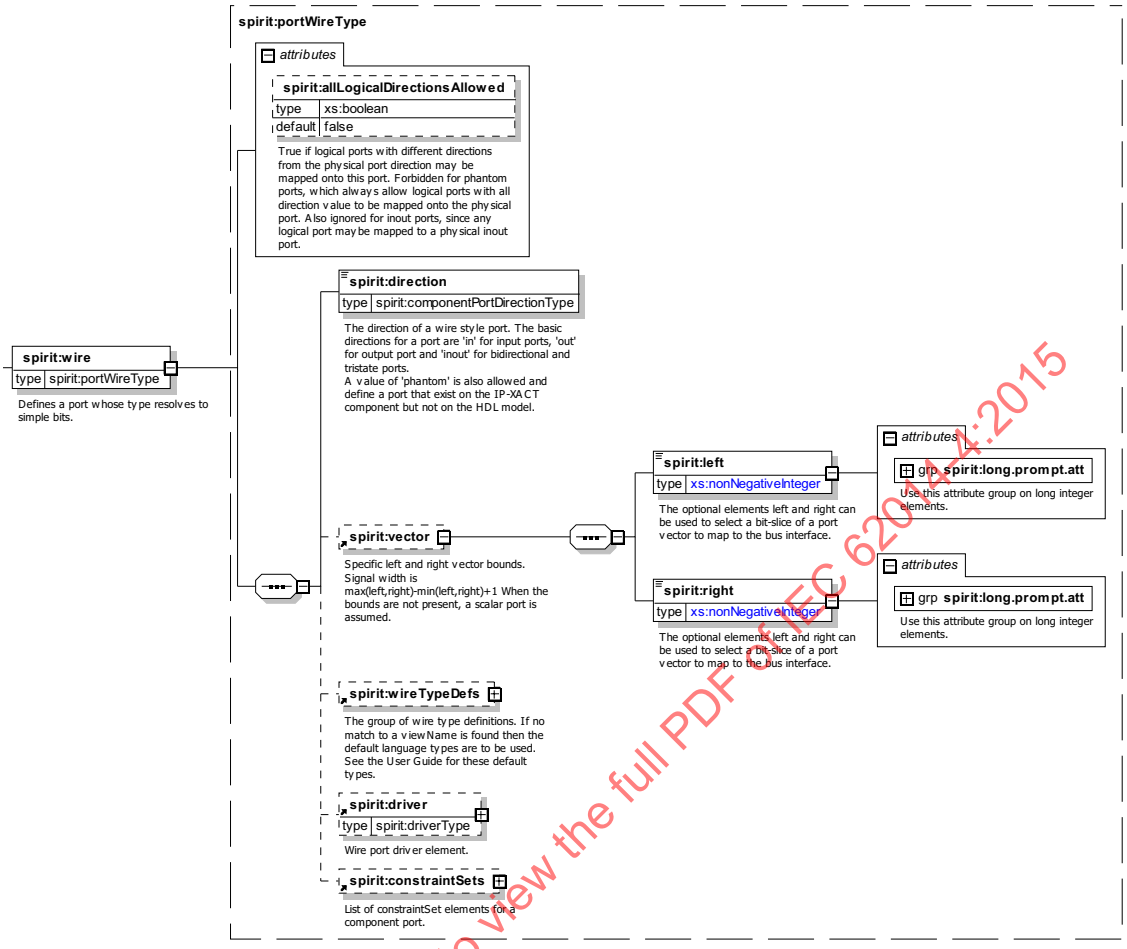
This example shows a component with a wire port (`clk`) and two transactional ports (`initiator` and `target`).

```
<spirit:ports>
   <spirit:port>
      <spirit:name>clk</spirit:name>
      <spirit:wire>
         <spirit:direction>in</spirit:direction>
      </spirit:wire>
   </spirit:port>
   <spirit:port>
      <spirit:name>initiator</spirit:name>
      <spirit:transactional>
         <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:name>initiator</spirit:name>
      <spirit:transactional>
         <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>
```

6.11.4 Component wire ports

6.11.4.1 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.



6.11.4.2 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. Scalar types in VHDL also include integer and enumeration values; however, scalars in IP-XACT only include binary values that relate to a single wire in a hardware implementation.

A wire port transports purely binary values or vectors of 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 whether the port may be mapped to a port in an **abstractionDefinition** with a different direction. The default value is **false**. The **allLogicalDirectionsAllowed** attribute is of type ***boolean***. See 5.3.

b) **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 that only exists on the IP-XACT component, but not on the implementation referenced from the view.

c) **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 C.12.

The **left** element means first boundary, the **right** element, the second boundary. **left** may be larger than **right** and **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*.

d) **wireTypeDefs** (optional) describes the ports type as defined by the implementation, see 6.11.5.

e) **driver** (optional) defines a driver that 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 6.11.6

f) **constraintSets** (optional) defines multiple set of constraints on a port used for synthesis or other operations. See 6.11.11.

See also: SCR 6.5, SCR 6.6, SCR 6.7, and SCR 6.12.

6.11.4.3 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
```

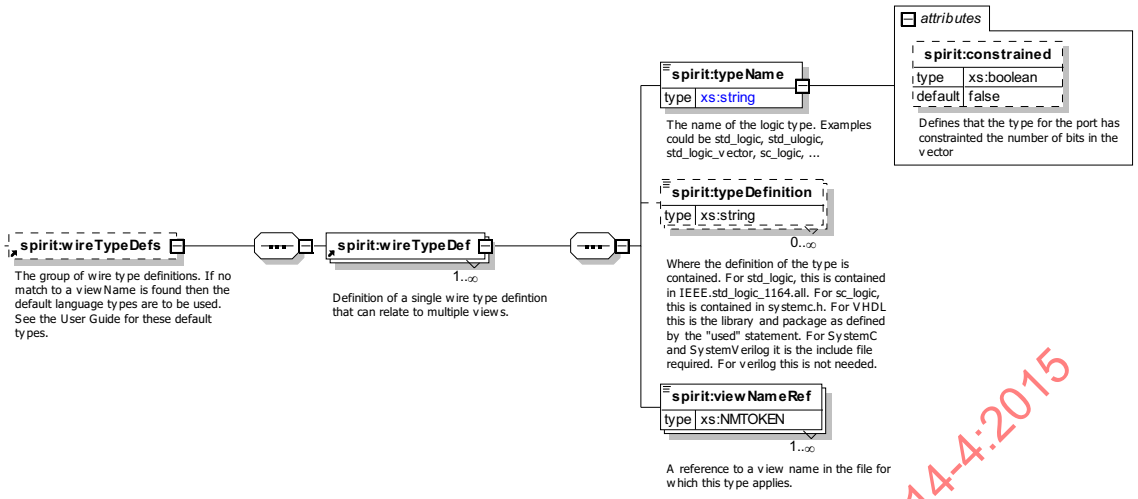
would be defined in IP-XACT as `left=29` and `right=3` with all bits in descending order.

```
<spirit:wire>
   <spirit:direction>out</spirit:direction>
   <spirit:vector>
      <spirit:left>29</spirit:left>
      <spirit:right>3</spirit:right>
   </spirit:vector>
</spirit:wire>
```

6.11.5 Component wireTypeDef

6.11.5.1 Schema

The following schema details the information contained in the **wireTypeDef** element, which may appear as an element inside the **wire** element of a 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.



6.11.5.2 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`. The **typeName** element is of type ***string***.

constrained (optional) attribute indicates whether or not the number of bits in the type declaration is fixed or not. The **constrained** attribute is of type ***boolean***. If the number of bits is fixed (**constrained == true**) the bit indices are not required when referencing the type. When **constrained** is **false** (the default), bit indices are required on all references to the type definition. See 6.11.5.2.1 and 6.11.5.2.2.

b) **typeDefinition** (optional) contains a language-specific reference to where the given type is actually defined. Table 4 shows some examples. There can be multiple **typeDefinition**s for each port. The **typeDefinition** element is of type ***string***.

Table 4—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) indicates the view or views in which this type definition applies. Multiple views can use the same set of type properties by specifying multiple **viewNameRef** elements. The **viewNameRef** shall match a **view/name** in the containing object. The **viewNameRef** element is of type ***NMTOKEN***. See 6.11.2.

6.11.5.2.1 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 description contains the indices in XML to designate the width so the following types 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>
```

6.11.5.2.2 Unconstrained array type

An *unconstrained array type* is a type for which the indices of the array have not been specified in the definition, for example,

```
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>
```

6.11.5.2.3 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 5. For those languages not shown here, no defaults can be presumed.

Table 5—View defaults

Language	Single bit	Vectors
VHDL	`std_logic`	`std_logic_vector`
Verilog	`wire`	`wire`
SystemC	`sc_logic`	`sc_lv`
SystemVerilog	`logic`	`logic`

6.11.5.2.4 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 (see Table 5).

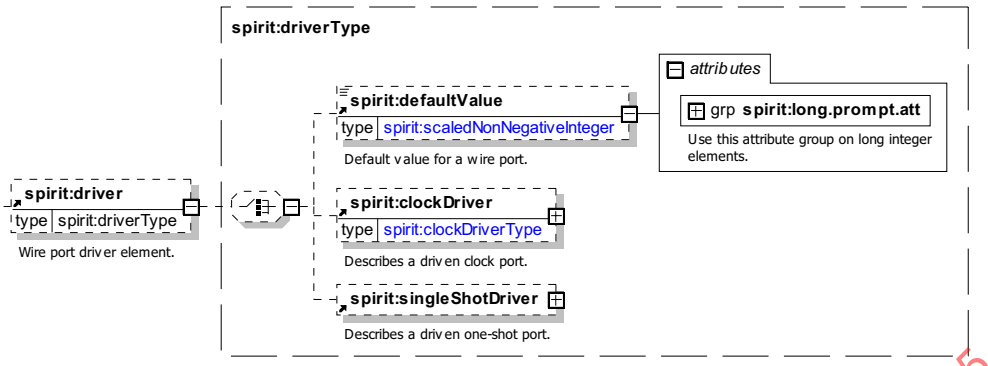
6.11.5.3 Example

See the examples in 6.11.5.2.2.

6.11.6 Component driver

6.11.6.1 Schema

The following schema details the information contained in the **driver** element, which may appear as an element inside the **wire** element of a 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; it is only allowed on ports with the direction **in** or **inout**.



6.11.6.2 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 value shall be applied to this port when it is left unconnected, independent of being part of a **busInterface**. This value shall be over-ridden by the default value from the abstraction definition if the interface is connected. The **defaultValue** element is of type ***scaledNonNegativeInteger***. The **defaultValue** element is configurable with attributes from ***long.prompt.att***, see C.12.

b) **clockDriver** (optional) specifies a repeating waveform for this port. See 6.11.7.

c) **singleShotDriver** (optional) specifies a non-repeating waveform for this port. See 6.11.8.

See also: SCR 6.26 and SCR 12.13.

6.11.6.3 Example

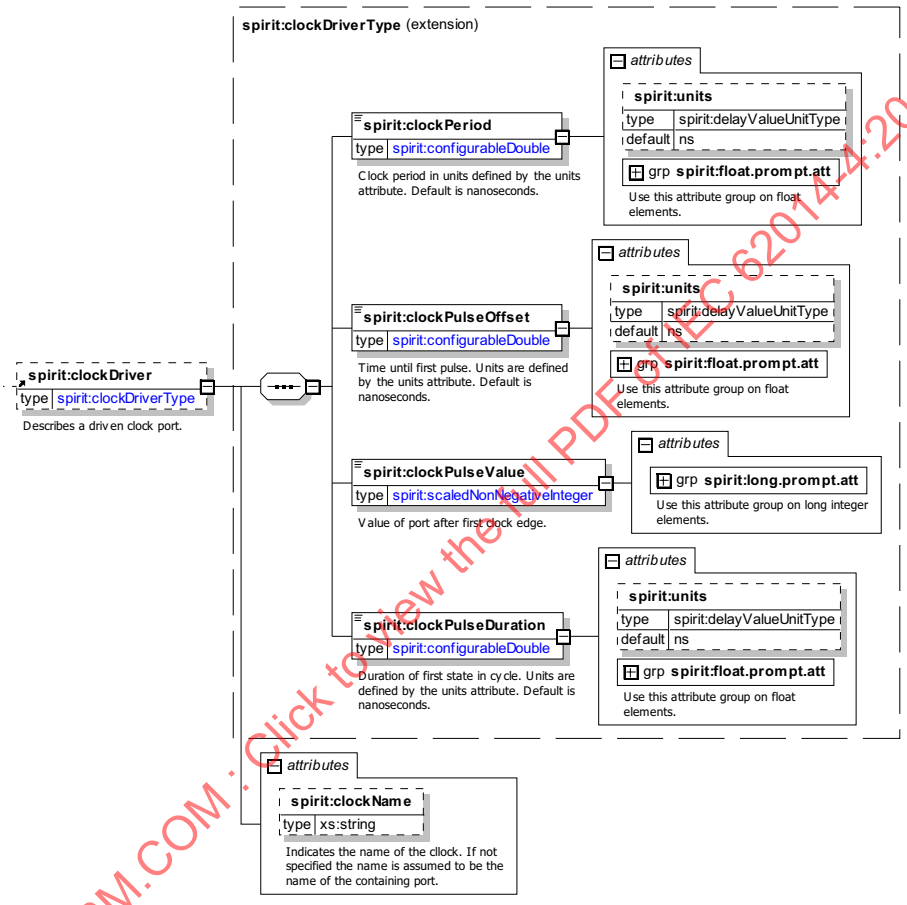
This example shows a default value of `0x0F` set for a vectored wire port named `scaler`.

```
<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>
```

6.11.7 Component driver/clockDriver

6.11.7.1 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**/**wire**/**driver** element. This element defines the properties of a clock waveform. When this element is contained within a non-scalar wire port, the clock waveform shall apply to all bits of this port.



6.11.7.2 Description

The **clockDriver** element contains four elements that describe the properties of a clock waveform. These are also depicted in Figure 11.

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 C.12. This element also contains a **units** (optional) attribute for specifying the units of the 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 C.12. This element also contains a **units** (optional) attribute for specifying the units of the time values: **ns** (the default) and **ps**.

c) **clockPulseValue** (mandatory) specifies the logic value to which the port transitions. This value is also the opposite of the value from which the waveform starts. The value of this element shall be `0` or `1`. The **clockPulseValue** element is of type ***scaledNonNegativeInteger***. The **clockPulseValue** element is configurable with attributes from ***long.prompt.att***, see C.12.

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 C.12. This element also contains a **units** (optional) attribute for specifying the units of the time values: **ns** (the default) and **ps**.

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. The **clockName** element is of type ***string***.

See also: SCR 12.9 and SCR 12.10.

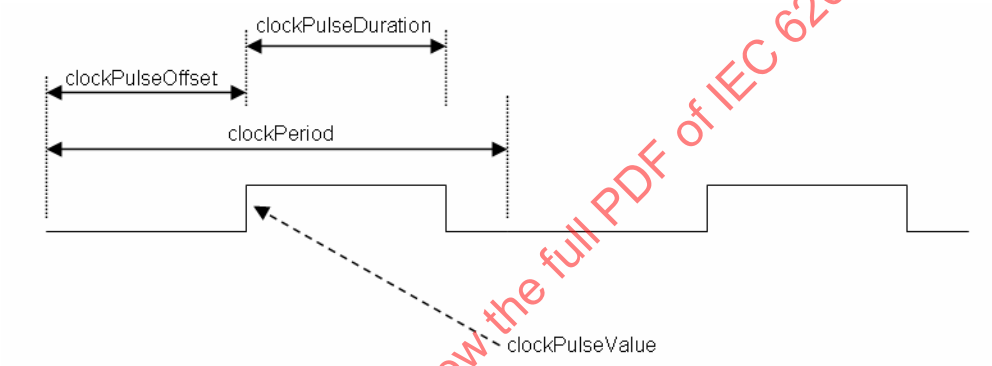


Figure 11—clockDriver elements

6.11.7.3 Example

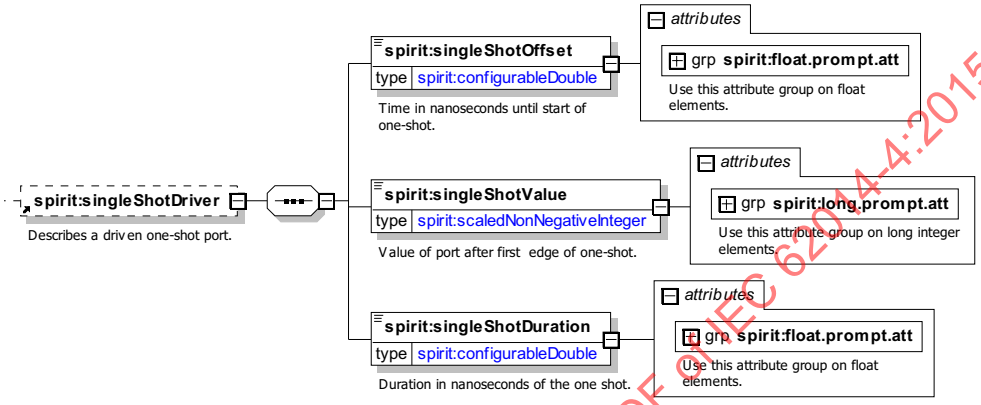
This is an example of a clock driver set on the wire port named `clk`. The clock starts off in the `logic 0` state for `4 ns`, then transitions to the `logic 1` state for `4 ns`. This cycle is repeated forever.

```
<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>
```

6.11.8 Component driver/singleShotDriver

6.11.8.1 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**/**wire**/**driver** element. This element defines the properties of a single-shot waveform. When this element is contained within a non-scalar wire port, the single-shot waveform shall apply to all bits of this port.



6.11.8.2 Description

The **singleShotDriver** element contains three elements that describe the properties of the waveform. These are also depicted in Figure 12.

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 C.12. This element also contains a **units** (optional) attribute for specifying the units of the 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 port transitions. This value is also the opposite of the value from which the waveform starts. The value of this element shall be `0` or `1`. The **singleShotValue** element is of type ***scaledNonNegativeInteger***. The **singleShotValue** element is configurable with attributes from ***long.prompt.att***, see C.12.

c) **singleShotDuration** (mandatory) specifies how 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 C.12. This element also contains a **units** (optional) attribute for specifying the units of the time values: **ns** (the default) and **ps**.

See also: SCR 12.11 and SCR 12.12.

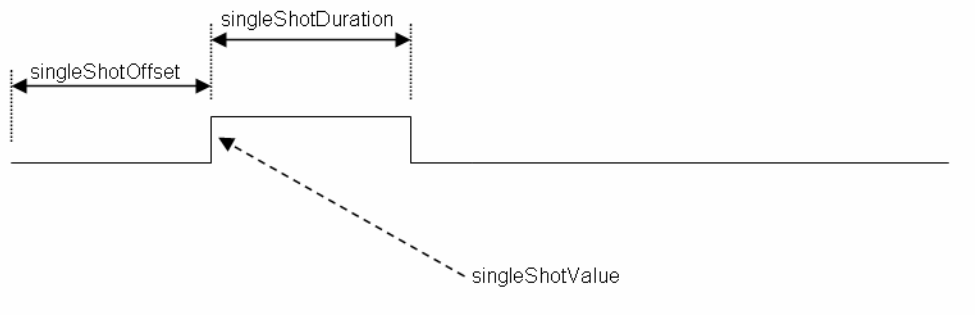


Figure 12—singleShotDriver elements

6.11.8.3 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 for `100 ns`.

```
<spirit:port>
   <spirit:name>reset</spirit:name>
   <spirit:wire>
      <spirit:direction>in</spirit:direction>
      <spirit:driver>
         <spirit:singleShotDriver>
            <spirit:singleShotOffset>100</spirit:singleShotOffset>
            <spirit:singleShotValue>1</spirit:singleShotValue>
            <spirit:singleShotDuration>100</spirit:singleShotDuration>
         </spirit:singleShotDriver>
      </spirit:driver>
   </spirit:wire>
</spirit:port>
```

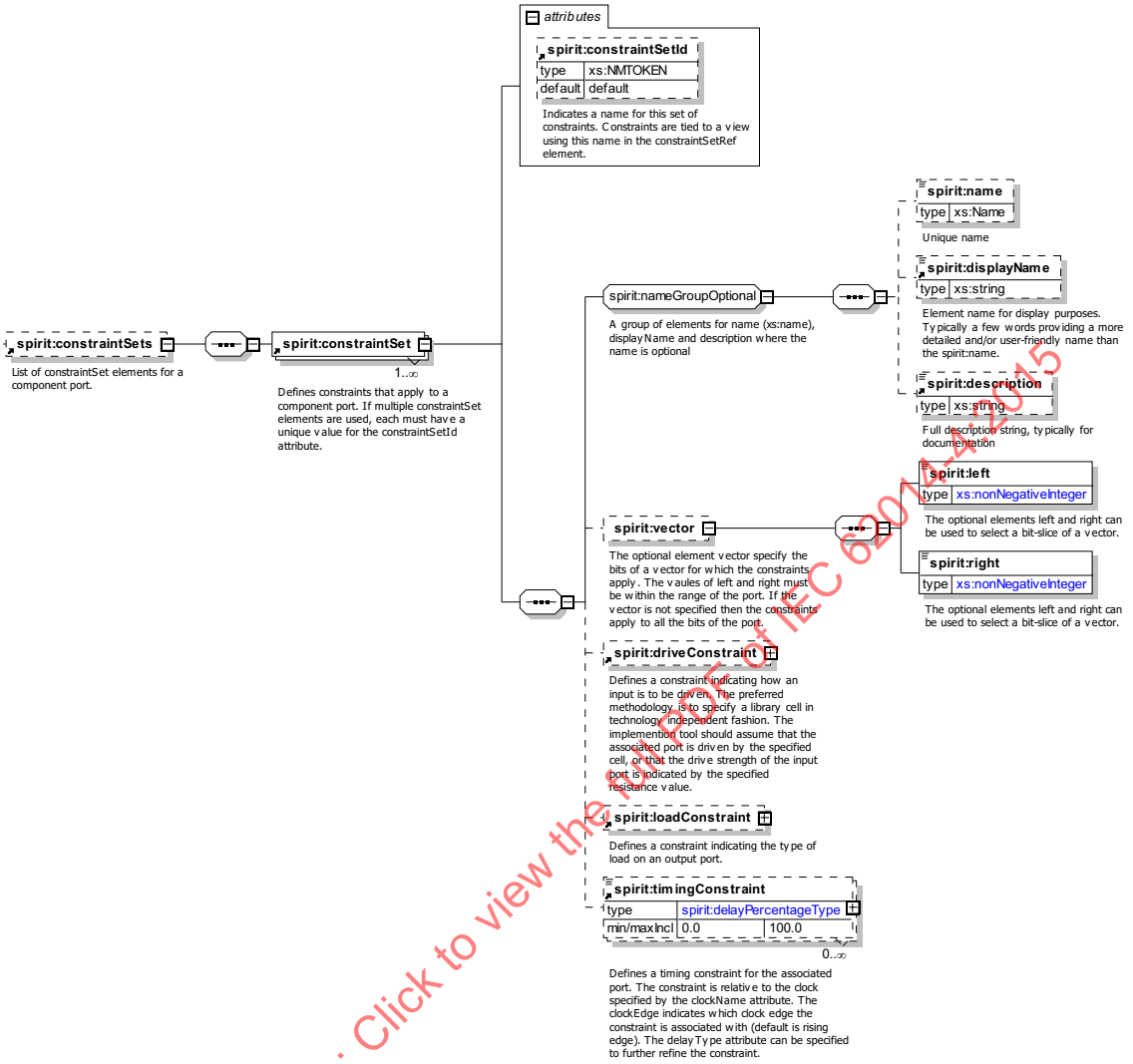

6.11.9 Implementation constraints

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.

6.11.10 Component wire port constraints

6.11.10.1 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**).



6.11.10.2 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 that 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** contains the following elements.

a) ***nameGroupOptional*** is defined in C.2.

b) **vector** (optional) determines to which bits of a vectored port the constraint applies. The **left** and **right** vector bounds elements inside the **vector** element specify the bounds of the vector. The **left** and **right** elements are of type ***nonNegativeInteger***.

c) **driveConstraint** (optional) defines a drive constraint for this port. See 6.11.11 for details.

d) **loadConstraint** (optional) defines a load constraint for this port. See 6.11.12 for details.

e) **timingConstraint** (optional) defines a timing constraint relative to a clock for this port. See 6.11.13 for details.

NOTE—To specify technology-dependent constraints (which is not represented in the schema), use an SDC file and reference the file via **fileSet**.

6.11.10.3 Example

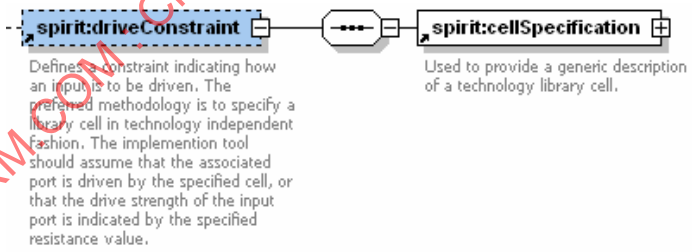
This example shows a port containing a single timing constraint appearing in two different constraint sets.

```
<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>
```

6.11.11 Port drive constraints

6.11.11.1 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.



6.11.11.2 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 6.11.14).

The **driveConstraint** element is not valid on an output port.

See also: SCR 12.1, SCR 12.3, and SCR 12.6.

6.11.11.3 Example

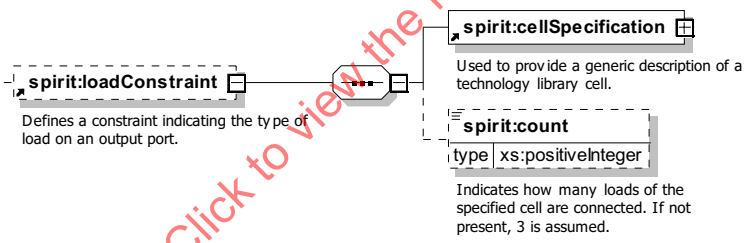
This example shows two different drive constraints. The first represents a median-strength D flop and the second a low-strength sequential cell.

```
<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>
```

6.11.12 Port load constraints

6.11.12.1 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.



6.11.12.2 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 6.11.14).

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 12.2, SCR 12.4, and SCR 12.5.

6.11.12.3 Example

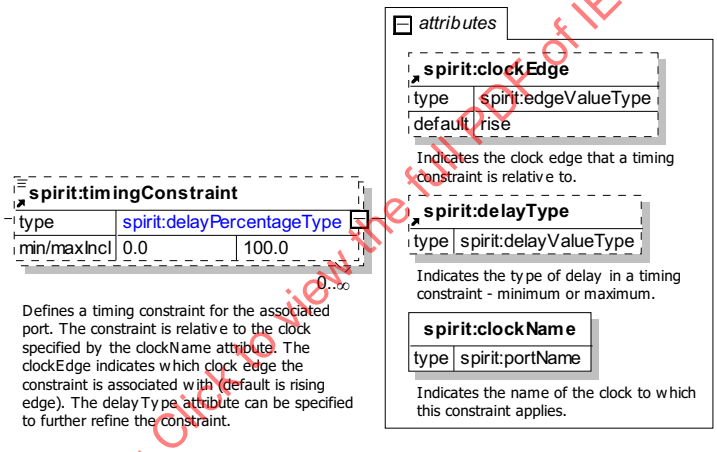
This example shows two different load constraints. The first is load consisting of three D flops of median strength and the second is a load consisting of four low-strength sequential cells.

```
<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>
```

6.11.13 Port timing constraints

6.11.13.1 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.



6.11.13.2 Description

The **timingConstraint** element defines a technology-independent timing constraint associated with the containing wire port of a component or abstraction definition. It is 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 that 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. **clockName** shall be a valid port name or another clock name in the containing description. 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 ***portName***.

See also: SCR 12.7 and SCR 12.8.

6.11.13.3 Example

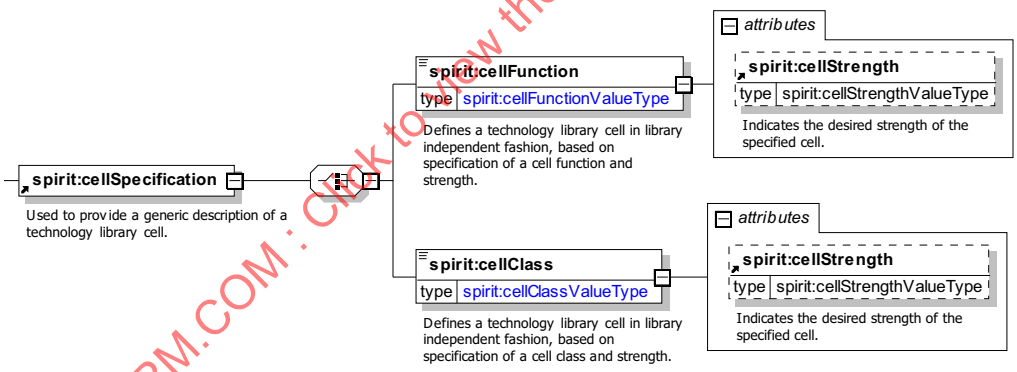
This example shows three basic timing constraints. The first indicates a delay of 40% of the clock `hclk`, relative to the rising edge of `hclk`, and applicable to both best- and worst-case timing analysis. The second indicates a delay of 30% of the clock `hclk`, relative to the falling edge of `hclk`, 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>
```

6.11.14 Load and drive constraint cell specification

6.11.14.1 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.



6.11.14.2 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 DE can map it to an appropriate cell in a specific library when the actual constraint is generated. The **cellSpecification** element shall 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.

6.11.14.3 Example

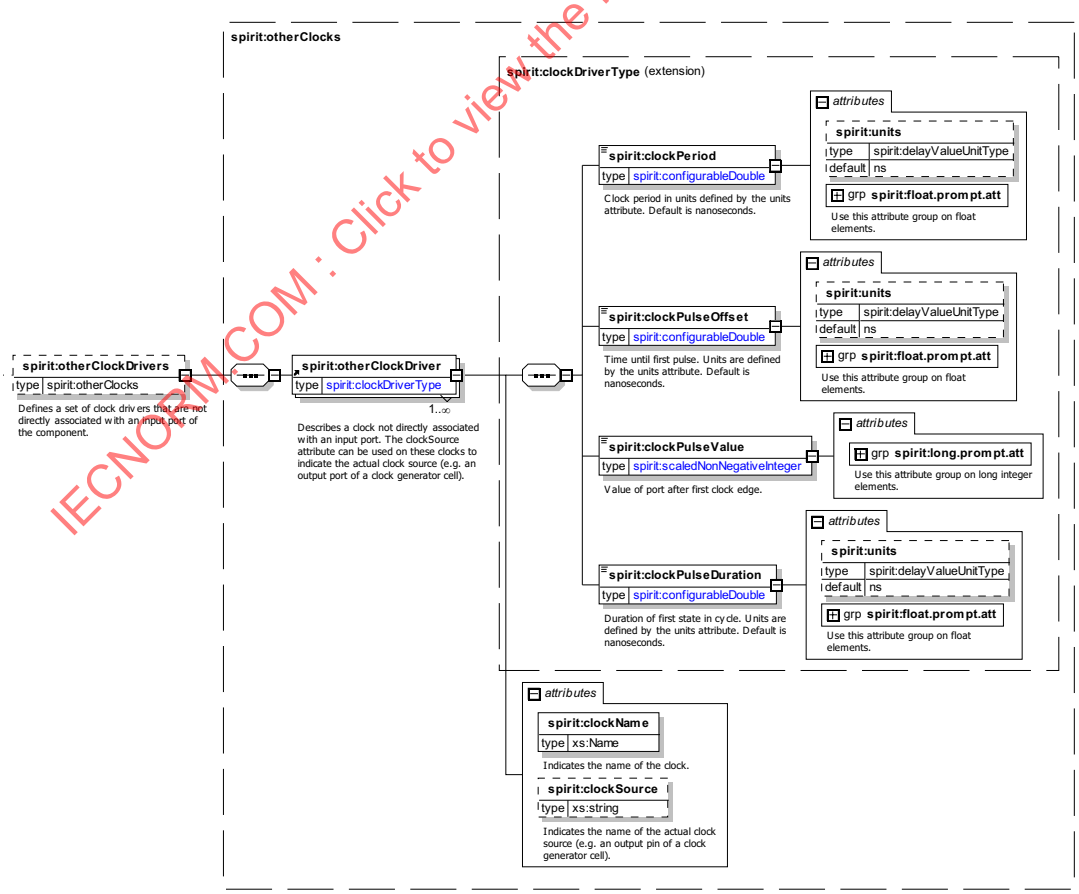
This example shows two different variations of cell specifications. The first indicates a median-strength D flop cell and the latter a low-strength sequential cell.

```
<spirit:cellSpecification>
   <spirit:cellFunction>dff</spirit:cellFunction>
</spirit:cellSpecification>
<spirit:cellSpecification>
   <spirit:cellClass spirit:strength=”low”>sequential</spirit:cellClass>
</spirit:cellSpecification>
```

6.11.15 Other clock drivers

6.11.15.1 Schema

The following schema defines the information contained in the **otherClockDrivers** element, which may appear within a **component** element.



6.11.15.2 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 subelements that 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 6.11.7.

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 of the clock generation cell. The **clockSource** element is of type ***string***.

6.11.15.3 Example

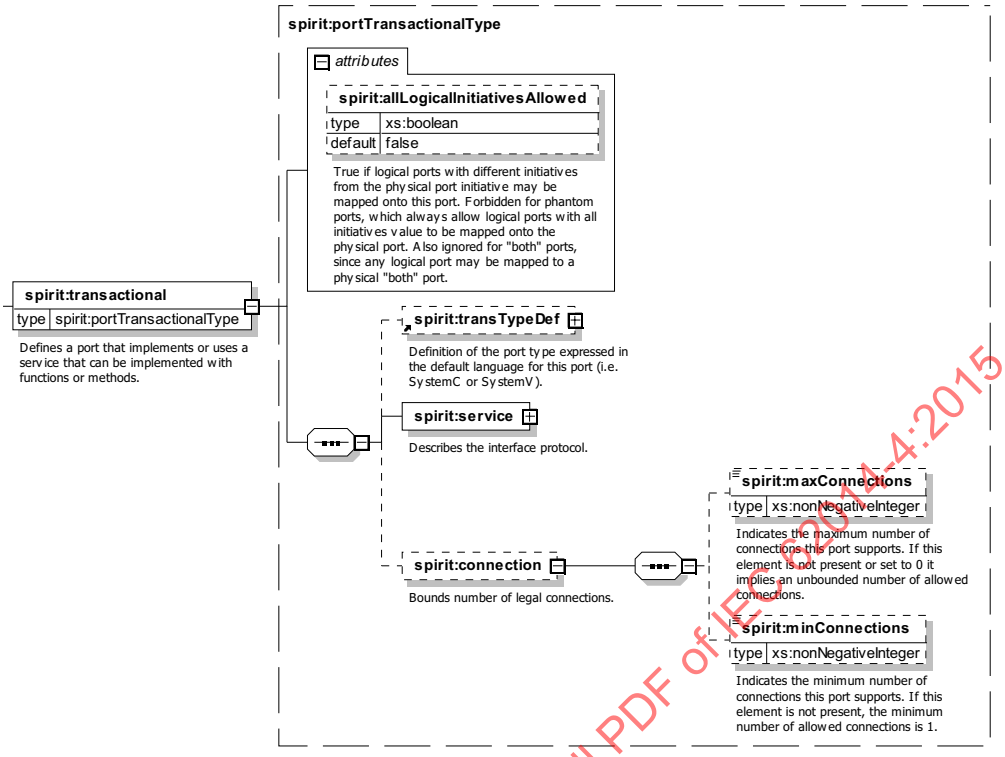
This example shows a virtual and a generated clock within the **otherClockDrivers** element.

```
<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>
```

6.11.16 Component transactional port type

6.11.16.1 Schema

The following schema defines the information contained in the **transactional** element (in a **component**/**model**/**ports**/**port** element).



6.11.16.2 Description

A **transactional** element in a component model port defines 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) **allLogicalInitiativesAllowed** (optional) attribute defines whether the port may be mapped to a port in an **abstractionDefinition** with a different initiative. The default value is **false**. The **allLogicalInitiativesAllowed** attribute is of type ***boolean***. See 5.3.

b) **transTypeDef** (optional) defines the port type expressed in the default language for this port. See 6.11.17.

c) **service** (mandatory) describes the interface protocol associated with the transactional port. See 6.11.18.

d) **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***.

See also: SCR 6.2, SCR 6.3, SCR 6.4, SCR 6.13, SCR 6.24, and SCR 6.25.

6.11.16.3 Example

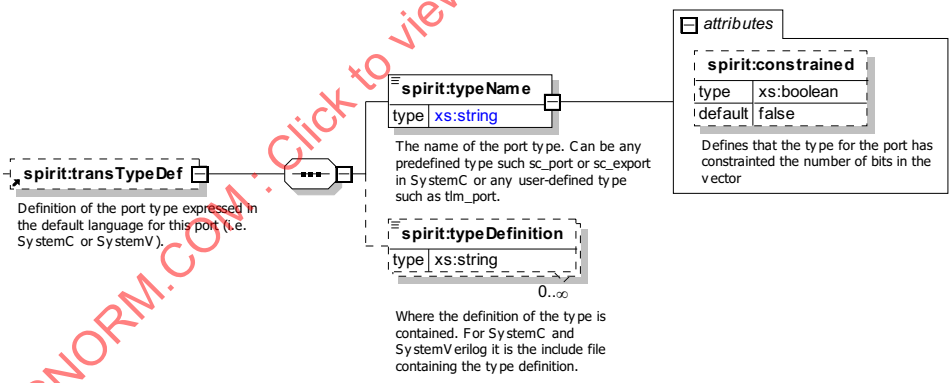
This example shows a transactional port requiring a service of type `tlm_interface` and allowing only a point-to-point connection.

```
<spirit:port> <spirit:name>tlm_initiator_port</spirit:name>
   <spirit:transactional>
      <spirit:transTypeDef>
         <spirit:typeName>sc_port</spirit:typeName>
      </spirit:transTypeDef>
   <spirit:service>
      <spirit:initiative>requires</spirit:initiative>
      <spirit:serviceTypeDefs>
         <spirit:serviceTypeDef>
            <spirit:typeName>tlm_interface</spirit:typeName>
         </spirit:serviceTypeDef>
      </spirit:serviceTypeDefs>
      </spirit:service>
         <spirit:connection>
            <spirit:maxConnections>1</spirit:maxConnections>
         </spirit:connection>

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

6.11.17 Component transactional port type definition

6.11.17.1 Schema

The following schema defines the information contained in the **transTypeDef** element (in a **component**/**model**/**ports**/**port**/**transactional** element).



6.11.17.2 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) contains a language-specific reference to where the given type is actually defined. Table 4 shows some examples. There can be multiple **typeDefinition**s for each port. The **typeDefinition** element is of type ***string***.

6.11.17.3 Example

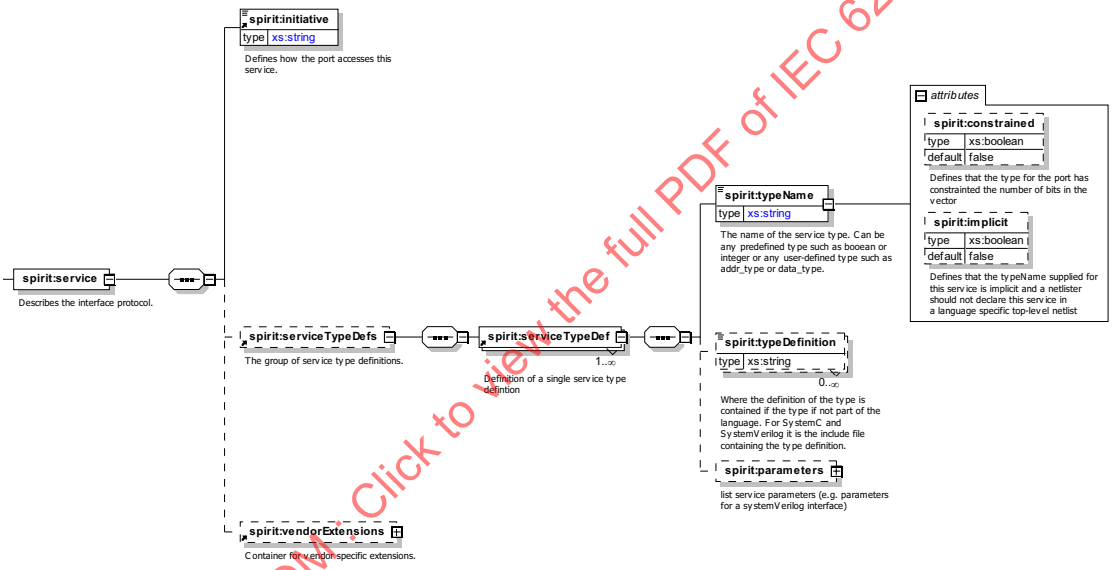
The following example shows the transactional type definition of a custom specific `tlm_port`, defined in the include file `tlm_port.h`.

```
<spirit:transTypeDef>
    <spirit:typeName>tlm_port</spirit:typeName>
    <spirit:typeDefinition>tlm_port.h</spirit:typeDefinition>
</spirit:transTypeDef>
```

6.11.18 Component transactional port service

6.11.18.1 Schema

The following schema defines the information contained in the **service** element (in a **component**/**model**/**ports**/**port**/**transactional** element).



6.11.18.2 Description

A **service** element describes the interface protocol associated with 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) **both** indicates the type of access is both **requires** and **provides**.

   3) **phantom** indicates a phantom port is being defined. See 6.11.19.

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 C.11.

c) **vendorExtensions** (optional) adds any extra vendor-specific data related to the service. See C.10.

6.11.18.3 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 spirit:name="addr" spirit:resolve="user">ADDR
               </spirit:parameter>
            <spirit:parameter spirit:name="data" spirit:resolve="user">DATA
               </spirit:parameter>
         </spirit:parameters>
      </spirit:serviceTypeDef>
   </spirit:serviceTypeDefs>
</spirit:service>
```

6.11.19 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.

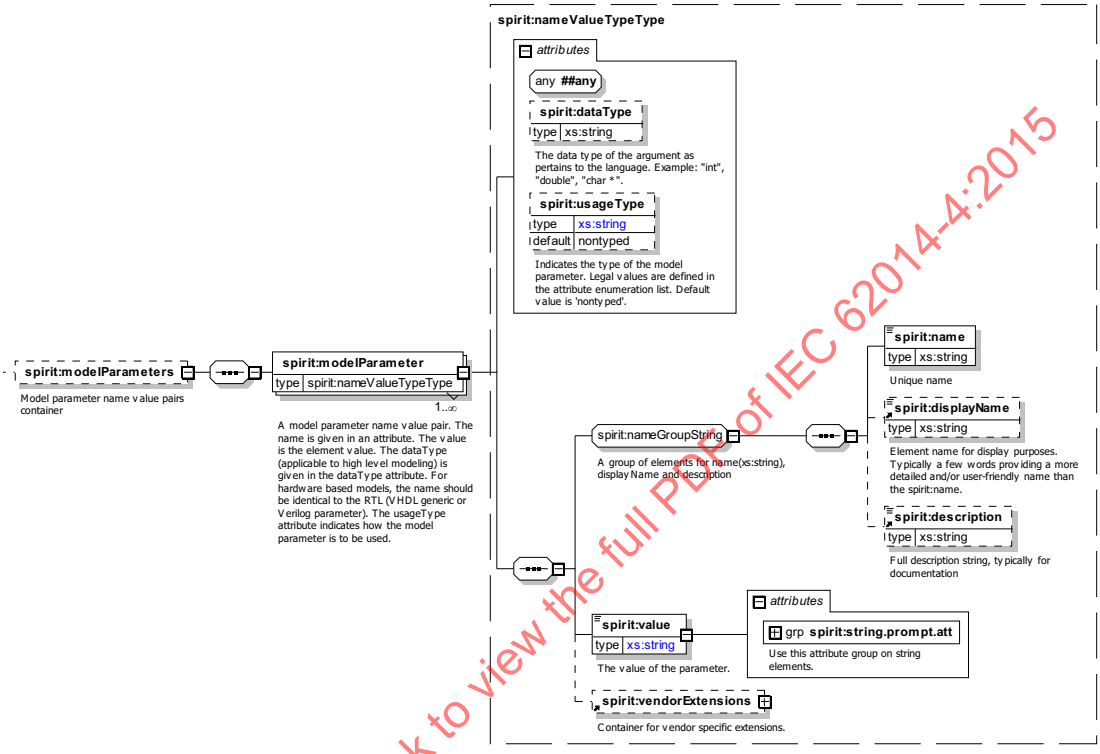
An IP-XACT channel is supposed to represent the complete bus infrastructure between the master, slave, and system bus interfaces connected to the bus. As such, the component containing the channel should contain everything that is needed to create this infrastructure. In many buses, however, some signals 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.

6.11.20 modelParameters

6.11.20.1 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.



6.11.20.2 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 of **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 as such has no IP-XACT semantic meaning. For example, SystemC could be `int`, `double`, `char*`, etc. For VHDL, this could be `std_logic`, `std_logic_vector`, `integer`, etc. The **dataType** attribute is of type ***string***.

b) **usageType** (optional) attribute specifies how this parameter is used in different modeling languages: **nontyped** (the default) and **typed**. See 6.11.20.2.1.

c) ***nameGroup*** group is defined in C.1.

d) **value** (mandatory) contains the actual value of the **modelParameter**. The **value** element is of type ***string***. The **value** element is configurable with attributes from ***string.prompt.att***, see C.12.

e) **vendorExtensions** (optional) adds any extra vendor-specific data related to the **modelParameter**. See C.10.

6.11.20.2.1 Typed and non-typed parameters classification

There are two categories of parameters: typed and non-typed.

The *typed* 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>
```

6.11.20.2.2 Generic parameters mapping in different languages

Table 6 summarizes the two kinds of parameters (initialization and declaration) expressed in the four most commonly used hardware languages.

Table 6—Parameter mappings

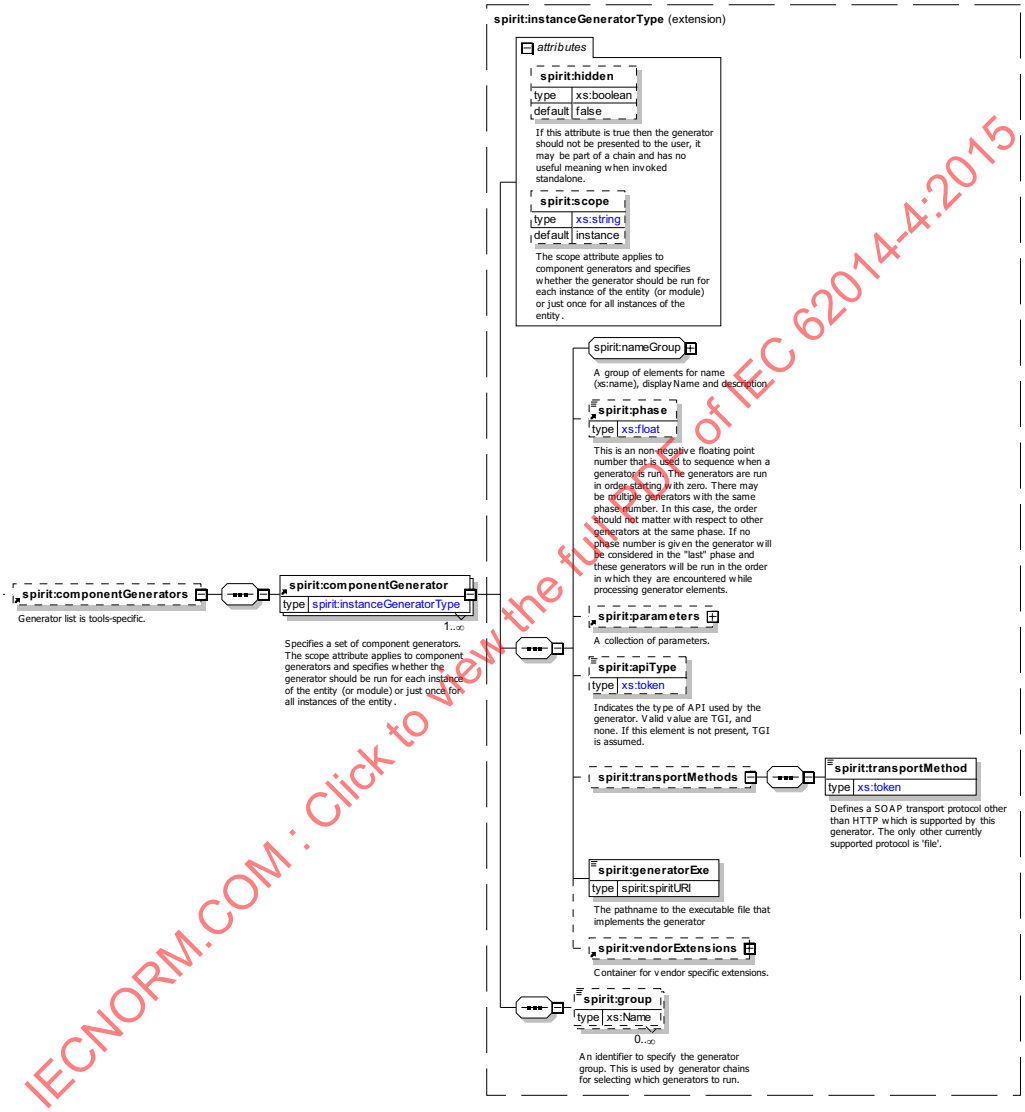
Language	Non-typed parameters (initialization)	Typed parameters (declaration)
VHDL	`generics`	NA
Verilog	`parameter`	NA
SystemC	`constructor`	`template` (`constant` or `variable` type)
SystemVerilog	`parameter`	`parameter`

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");`).

6.12 Component generators

6.12.1 Schema

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



6.12.2 Description

The **componentGenerators** element contains an unbounded list of **componentGenerator** elements. Each **componentGenerator** element defines a generator that is 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 a stand-alone generator and is required to be run as part of a chain. This generator should not be presented to the user for direct invocation. If **false** (the default), this generator may be run as a stand-alone 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 is defined in C.1. The **name** elements shall be unique within the containing **componentGenerators** element.

b) **phase** (optional) determines the sequence in which generators are run. Generators are run in order starting with zero (0). If two generators have the same phase number, 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 any order after the last generator with a phase number. The **phase** element is of type ***float*** and shall also be a positive number.

c) **parameters** (optional) specifies any **componentGenerator** parameters. See C.11.

d) **apiType** (optional) indicates the type of API used by the generator: an enumerated list of **TGI** or **none**. **TGI** indicates the generator communicates with the DE using SOAP as defined by the IP-XACT TGI. **none** indicates the generator does not communicate with the DE.

e) **transportMethods** (optional) defines alternate SOAP transport protocols that this generator can support. The default SOAP transport protocol is HTTP if this element is not present.

   1) **transportMethod** specifies an alternate transport protocol. This element is an enumerated list of only one element **file**.

   2) **file** indicates the SOAP transport protocol is transported to the DE via a file or file handle.

f) **generatorExe** (mandatory) contains an absolute or relative (to the location of the containing document) 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**. See C.11.

h) **group** (optional) is an unbounded 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 9.1. The **group** element is of type ***Name***.

6.12.3 Example

This example shows a component generator used to validate the connections to a component.

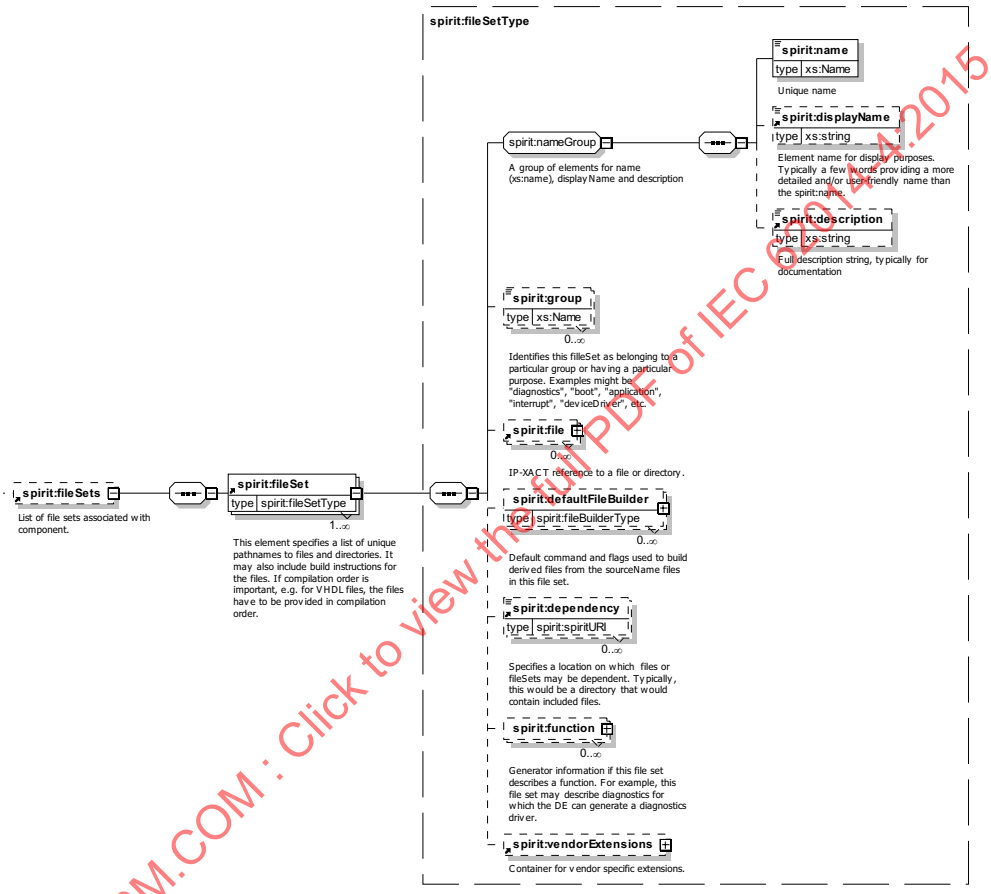
```
<spirit:componentGenerator>
   <spirit:name>connectionRuleChecker<spirit:name>
   <spirit:phase>100.0</spirit:phase>
   <spirit:parameters>
      <spirit:parameter>
         <spirit:name>language<spirit:name>
         <spirit:value spirit:id="checker" spirit:resolve="user">strict</
   spirit:value>
      </spirit:parameter>
   </spirit:parameters>
   <spirit:apiType>TGI</spirit:apiType>
   <spirit:generatorExe>../TGI/checker.tcl</spirit:generatorExe>
   <spirit:group>checker</spirit:group>
</spirit:componentGenerator>
```

6.13 File sets

6.13.1 fileSets

6.13.1.1 Schema

The following schema details the information contained in the **fileSets** element, which may appear in a component or an abstractor.



6.13.1.2 Description

The **fileSets** element contains one or more **fileSet** elements. A **fileSet** contains a list of files and directories associated with a component and/or instructions for further processing. If compilation order is important (e.g., for VHDL files), the files shall be listed in the order needed for compilation (the files to compile first are listed first). **fileSet** has the following mandatory and optional elements.

a) ***nameGroup*** group is defined in C.1. The **name** elements shall be unique within the containing **fileSets** element.

b) **group** (optional) describes the function or purpose of the file set with a single unbounded word group name (e.g., `diagnostics`, `interrupt`, etc.). The **group** element is of type ***Name***.

c) **file** (optional) references a single unbounded file or directory associated with the file set. If compilation order is important (e.g., for VHDL files), the files shall be listed in the order needed for compilation (see 6.13.2).

d) **defaultFileBuilder** (optional) specifies the unbounded default build commands for the files within this file set. See 6.13.5.

e) **dependency** (optional) 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) specifies the unbounded information about a software function for a generator (see 6.13.6).

g) **vendorExtensions** (optional) provides a place for any vendor-specific extensions. See C.10.

6.13.1.3 Example

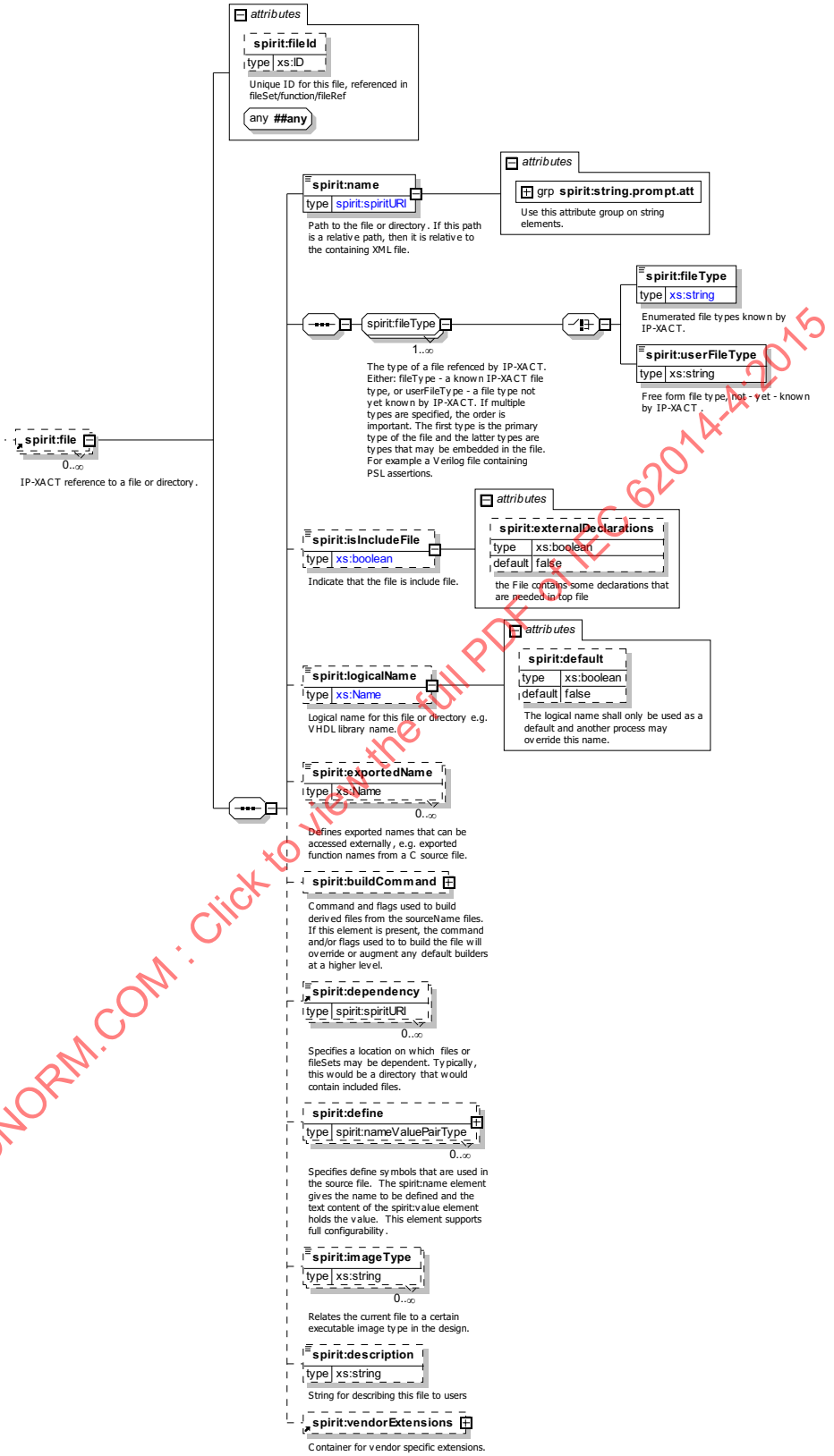
The following is an example of a **fileSet** with two VHDL files.

```
<spirit:fileSets>
    <spirit:fileSet
        <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>
```

6.13.2 file

6.13.2.1 Schema

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



6.13.2.2 Description

A **file** is a reference to a file or directory. It is an optional element of a **fileSet**. If compilation order is important (e.g., for VHDL files), the files shall be listed in the order needed for compilation (the files to compile first are listed first). The **file** element contains an attribute **fileId** (optional) that is used for references to this file from inside the **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 document) 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 (see D.17). The **name** element is of type ***spiritURI***. The **name** element is configurable with attributes from ***string.prompt.att***, see C.12.

b) ***fileType*** (mandatory) group contains one or more of the elements defined in C.9.

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 **external-Declarations** (optional); when **true**, this indicates the include file is needed by users of any files in this file set.

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 (when **true**) the logical name shall only be used as a default and another process may override this name. If **false** (the default), this logical name shall always be used. The **default** attribute is of type ***boolean***.

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 **defaultFile-Builder** elements. See 6.13.3.

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) specifies the define symbols to use in the source file. See 6.13.4.

i) **imageType** (optional, unbounded) relates the current file to an executable image type in the design. The **imageType** element is of type ***string***.

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 C.10.

See also SCR 14.1.

6.13.2.3 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.

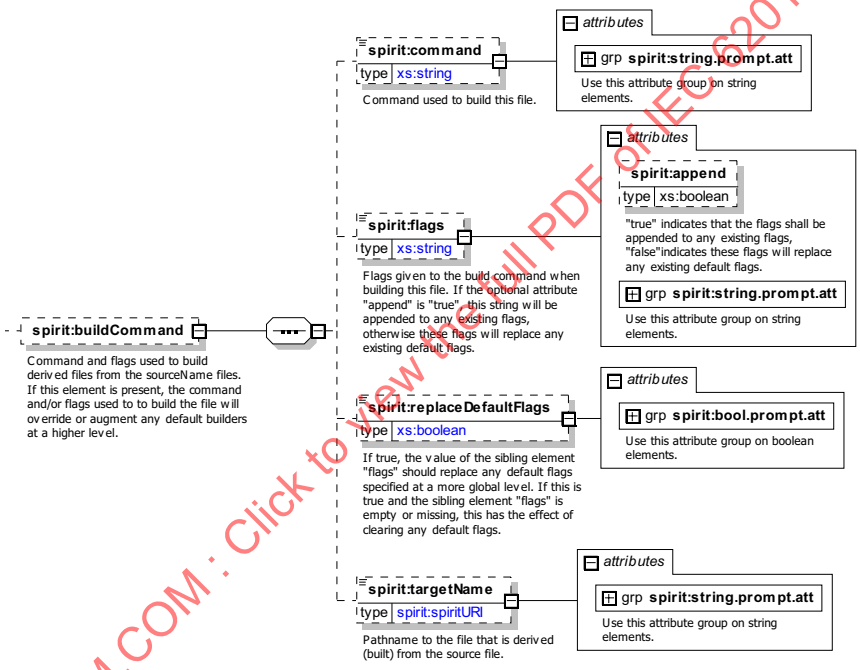
```
<spirit:fileSets>
   <spirit:fileSet>
      <spirit:name>fs-verilogSource</spirit:name>
      <spirit:file>
         <spirit:name>data/i2c/RTL/i2c.v</spirit:name>
         <spirit:fileType>verilogSource</spirit:fileType>
         <spirit:logicalName>i2c_lib</spirit:logicalName>
      </spirit:file>
      <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>
```

6.13.3 buildCommand

6.13.3.1 Schema

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



6.13.3.2 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 files of this type. The **command** element is of type ***string***. The **command** element is configurable with attributes from ***string.prompt.att***, see C.12.

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 C.12. The **flags** element contains an attribute **append** (optional), which when **true,** indicates the **flags** shall be appended to the current flags from the **defaultFileBuilder** (see 6.13.5), **fileBuilder** (see 6.7.5), or the build script generator. If **false**, the **flags** shall replace the existing flags.

c) **replaceDefaultFlags** (optional), when **true**, documents flags that replace any of the default flags from the build script generator. If **false**, the flags are appended. If **true** and the flags element is

empty or not present, this has the effect of clearing all the flags. If this element is not present, its effective value is **false**. The **replaceDefaultFlags** element is of type ***boolean***. The **replaceDefaultFlags** element is configurable with attributes from ***bool.prompt.att***, see C.12.

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 C.12.

6.13.3.3 Example

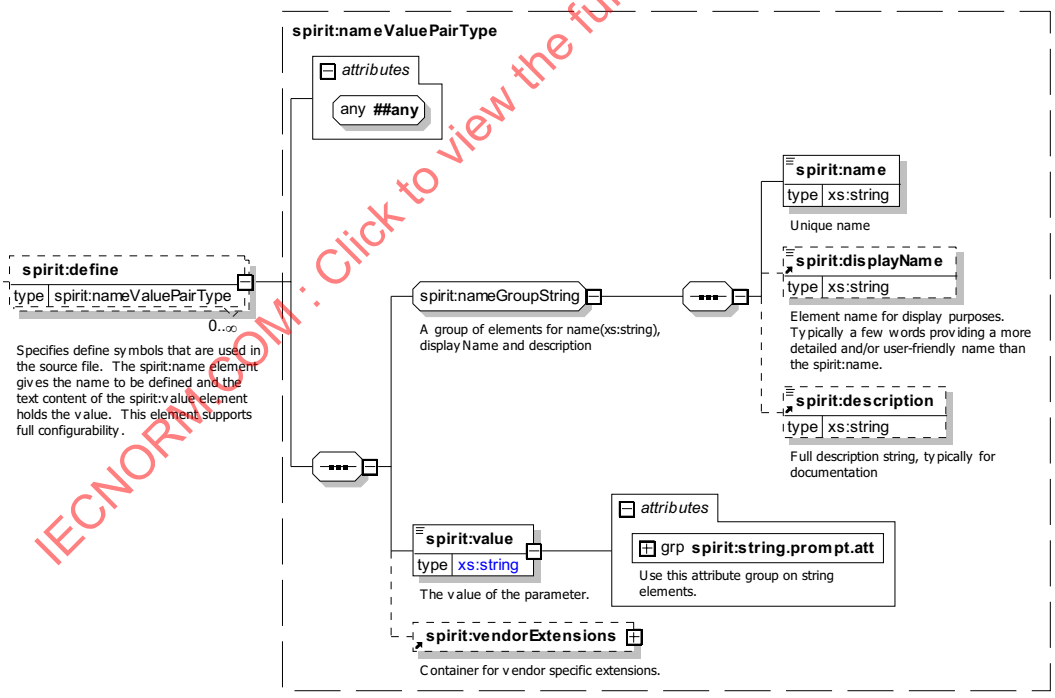
The following example specifies the build command for the containing file.

```
<spirit:buildCommand>
   <spirit:command>g++</spirit:command>
   <spirit:flags>-O</spirit:flags>
   <spirit:targetName>compiled/model.o<spirit:targetName>
</spirit:buildCommand>
```

6.13.4 define

6.13.4.1 Schema

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



6.13.4.2 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 is defined in C.5.

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 C.12.

c) **vendorExtensions** (optional) provides a place for any vendor-specific extensions. See C.10.

6.13.4.3 Example

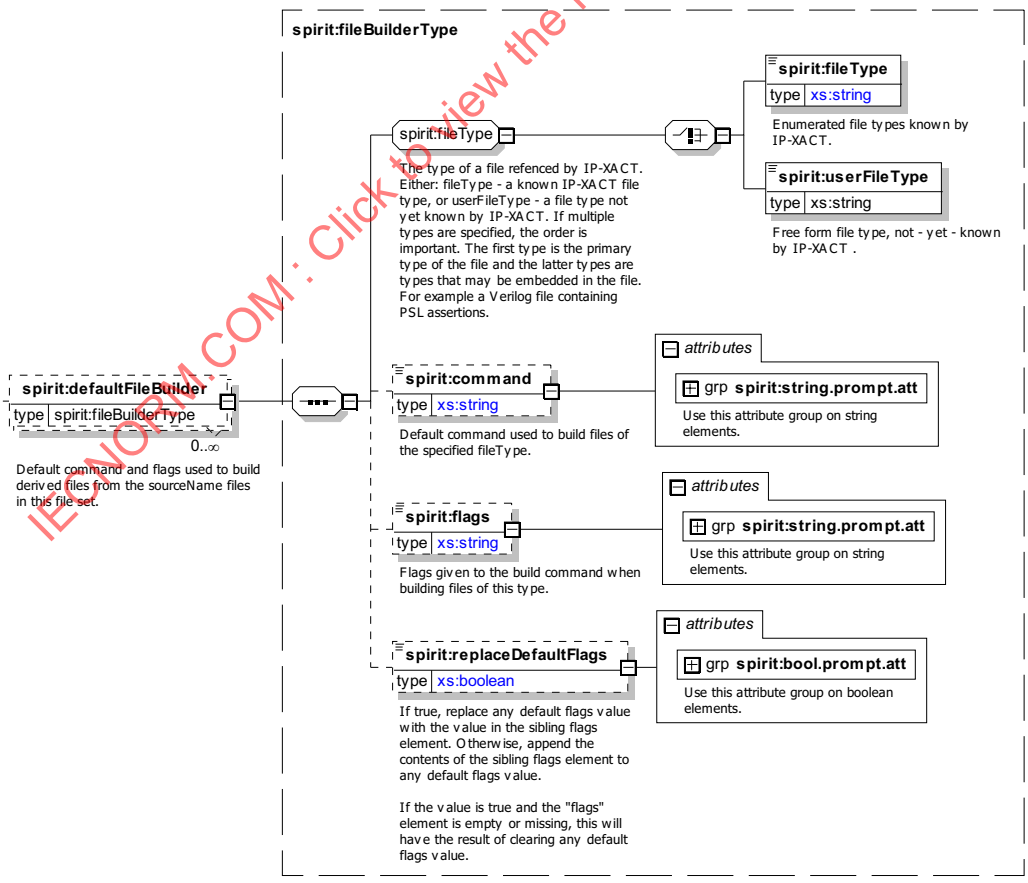
This example defines a symbol called `PROCESSOR_ARCH` to be equal to `amrV5`.

```
<spirit:define>
    <spirit:name>PROCESSOR_ARCH</spirit:name>
    <spirit:value>armV5</spirit:value>
</spirit:define>
```

6.13.5 defaultFileBuilder

6.13.5.1 Schema

The following schema details the information contained in the **defaultFileBuilder** element, which may appear as an element inside the **fileSet** or **view** element.



6.13.5.2 Description

A **defaultFileBuilder** contains default flags or commands for building the containing source file types. These flags or commands may be overridden by flags or commands present in lower-level **defaultFileBuilder** or **buildCommand** elements.

a) ***fileType*** (mandatory) group contains one or more of the elements defined in C.9.

b) **command** (optional) element defines a compiler or assembler tool that processes files of this type. The **command** element is of type ***string***. The **command** element is configurable with attributes from ***string.prompt.att***, see C.12.

c) **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 C.12.

d) **replaceDefaultFlags** (optional) when **true** indicates the **flags** shall be appended to the current flags. If **false**, the **flags** shall replace the existing flags. The **replaceDefaultFlags** element is of type ***boolean***. The **replaceDefaultFlags** element is configurable with attributes from ***bool.prompt.att***, see C.12.

6.13.5.3 Example

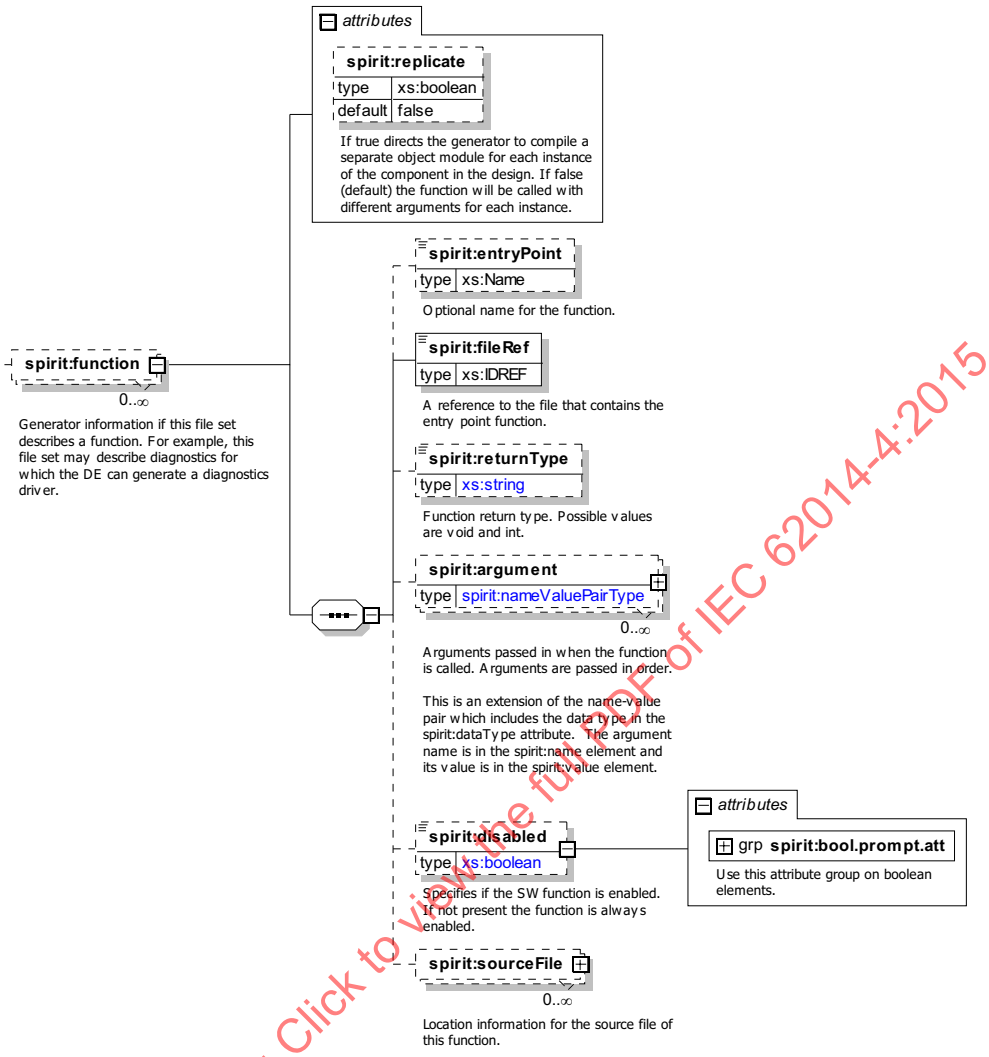
The following is an example that specifies the default compiler command to use.

```
<spirit:defaultFileBuilder>
    <spirit:fileType>cSource</spirit:fileType>
    <spirit:command>g++</spirit:command>
</spirit:defaultFileBuilder>
```

6.13.6 function

6.13.6.1 Schema

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



6.13.6.2 Description

A **function** specifies information about a software 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. The **entryPoint** element is of type ***Name***.

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**. The **fileRef** element is of type ***IDREF***. See 6.13.2.

c) **returnType** (optional) is an enumerated ***string*** type that indicates the return type for the function. The two possible values are **int** and **void**.

d) **argument** (optional) lists any arguments passed when this function is called. All arguments shall be passed in the order presented in this description. See 6.13.7.

e) **disabled** (optional) disables the software function. The **disabled** element is of type ***boolean***. When **true**, the software function is not available for use. When **false**, the function is available. If this element is not present, its effective value is **false**. The **disabled** element is configurable with attributes from ***bool.prompt.att***, see C.12.

f) **sourceFile** (optional) references any source files. The order of the source files may be important, as this could indicate a compile order. See 6.13.8.

6.13.6.3 Example

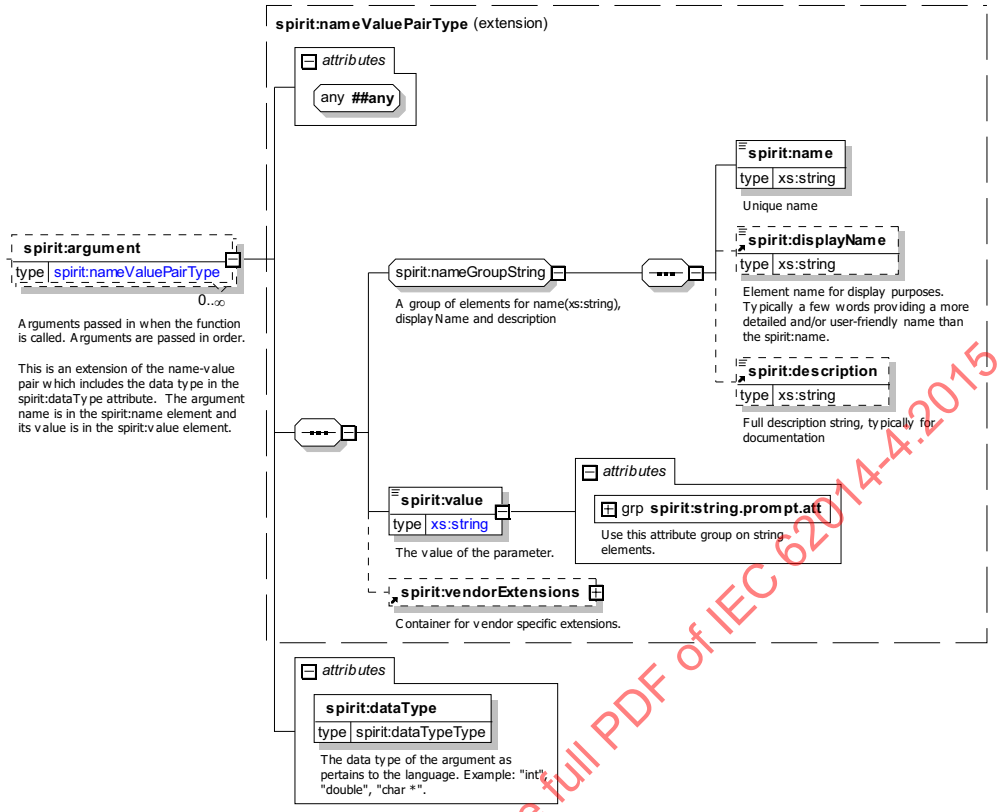
The following example includes a file with a `fileId` and a **function** referencing that file.

```
<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>
```

6.13.7 argument

6.13.7.1 Schema

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



6.13.7.2 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 is defined in C.5.

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 C.12.

c) **vendorExtensions** (optional) provides a place for any vendor-specific extensions. See C.10.

6.13.7.3 Example

The following example includes a file with a `fileId` and a **function** referencing that file.

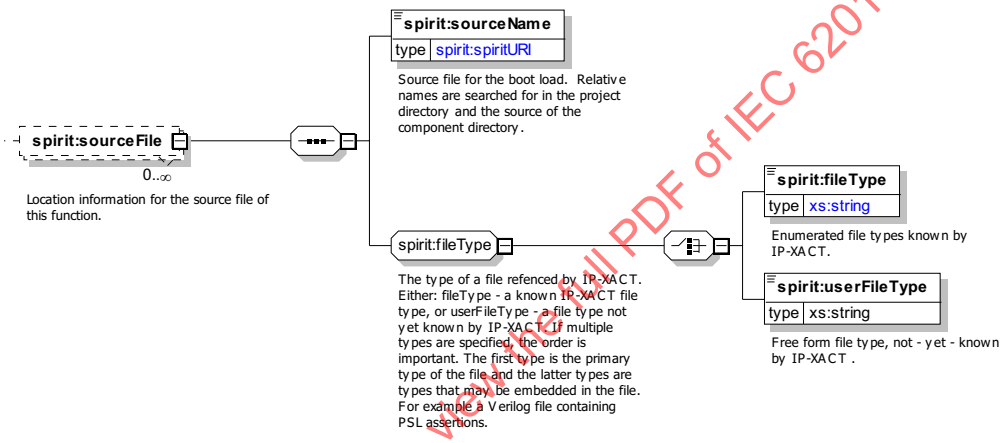
```
<spirit:fileSets>
   <spirit:fileSet spirit:fileSetId="fs-systemcSource">
      <spirit:name>sourceFiles</spirit:name>
      <spirit:file spirit:fileId="source">
         <spirit:name>src/source.cc</spirit:name>
         <spirit:fileType>systemCSource-2.1</spirit:fileType>
      </spirit:file>
      <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>
```

6.13.8 sourceFile

6.13.8.1 Schema

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



6.13.8.2 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 document) 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 **sourceName** element is of type ***spiritURI***.

b) ***fileType*** (mandatory) group contains one or more of the elements defined in C.9.

6.13.8.3 Example

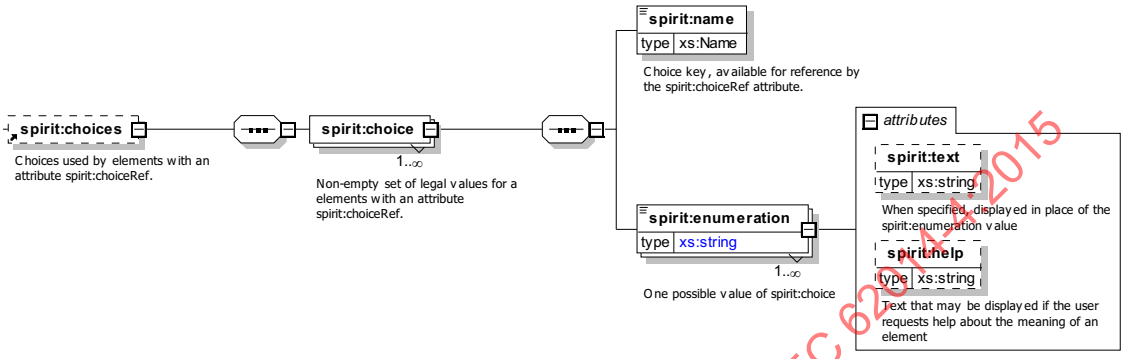
The following example specifies the type and location of a source file.

```
<spirit:source spirit:fileId="source">
    <spirit:sourceName>src/source.cc</spirit:sourceName>
    <spirit:fileType>systemCSource-2.1</spirit:fileType>
</spirit:source>
```

6.14 Choices

6.14.1 Schema

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



6.14.2 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 description.

The **choice** definition contains the following elements.

a) **name** (mandatory) specifies the name of this list and is used by other elements for reference. The **name** elements shall be unique within the containing **choices** element. The **name** element is of type ***Name***.

b) **enumeration** (mandatory) is an unbounded list of elements, where each holds a possible value that the referencing element may contain. The **enumeration** element is of type ***string***.

   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 5.11.

6.14.3 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:name>
```

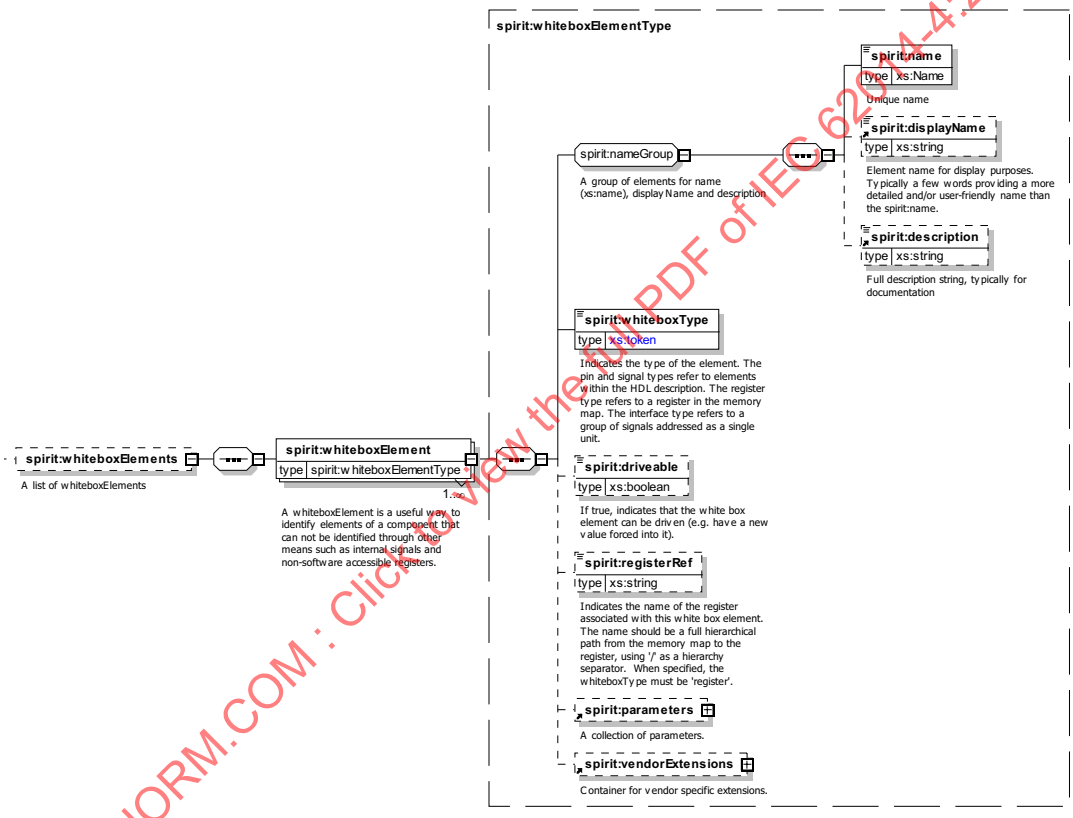
```
            <spirit:value spirit:format=”choice”
        spirit:choiceRef="widthOptions">1</spirit:value>
        </spirit:modelparameter>
        <spirit:modelparameter>
            <spirit:name>Dwidth</spirit:name>
            <spirit:value spirit:format=”choice”
        spirit:choiceRef="DwidthOptions">4</spirit:value>
        </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>
```

6.15 White box elements

Verification IP (VIP), with monitor bus interfaces, connect to an active bus interface to monitor only that interface's protocol for a variety of uses. Other verification tools may require access to component IP in a design, at a level deeper than the interfaces defined for the component. A white box element provides such access. This can be used in situations where internal registers, pins, signals, or whole IP-XACT interfaces need to be monitored or driven by VIP.

6.15.1 Schema

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



6.15.2 Description

The **whiteboxElements** element contains a list of one or more **whiteboxElement** elements. Each **whiteboxElement** element contains the following elements.

a) ***nameGroup*** group is defined in C.1. The **name** element shall be unique within the containing **whiteboxElements** element.

b) **whiteboxType** (mandatory) documents this white box element's referent: a **register**, **pin**, **signal**, or **interface** within the component. **register** indicates a register definition (referenced by the **registerRef** element) in this component can be mapped to physical signals. **pin** indicates a port on an internal instance in this component can be mapped to physical signals. **signal** indicates a signal between two internal instances in this component can be mapped to physical signals. **interface** indicates a group of signals that can be addressed as a single name.

In each case, the view-specific path is contained in the matching **model/view/whiteboxElementRef** element.

c) **drivable** (optional), when **true**, indicates the white box describes a point within the IP that can be driven, i.e., forced to a new value. If **false**, the white box references a point that cannot be driven. If this element is not present, its effective value is **false**. The **drivable** element is of type ***boolean***.

d) **registerRef** (optional) names the register indicated by this white box 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 white box that can be parameterized. See C.11.

f) **vendorExtensions** (optional) provides a space for any vendor-specific extensions. See C.10.

6.15.3 Example

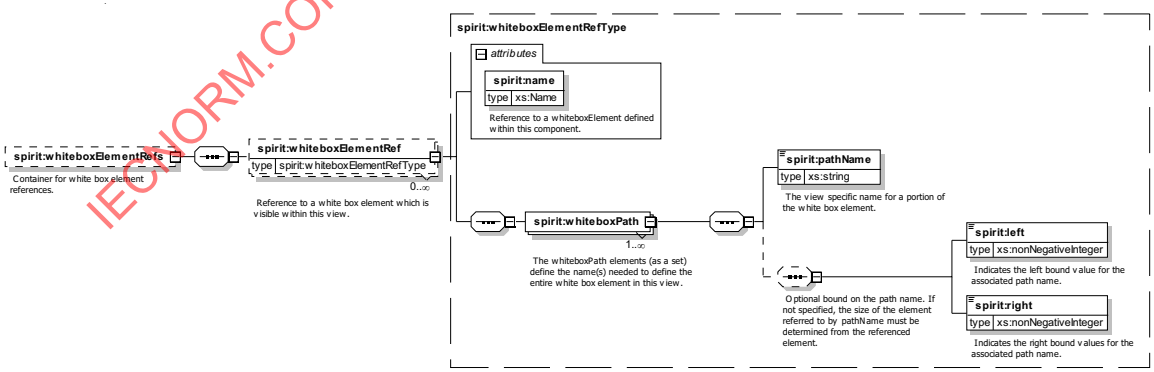
The following example shows the definition of a register (status) that can be accessed (i.e., during verification) within a component.

```
<spirit:whiteboxElements>
   <spirit:whiteboxElement>
      <spirit:name>Status</spirit:name>
      <spirit:whiteboxType>register</spirit:whiteboxType>
      <spirit:driveable>false</spirit:driveable>
      <spirit:registerRef>mmname/abname/status</spirit:registerRef>
   </spirit:whiteboxElement>
</spirit:whiteboxElements>
```

6.16 White box element reference

6.16.1 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.



6.16.2 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** attribute is of type ***Name***. **whiteboxElement** element contains the following elements.

whiteboxPath (mandatory) contains unbounded 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***.

See also SCR 12.14 and SCR 12.15.

6.16.3 Example

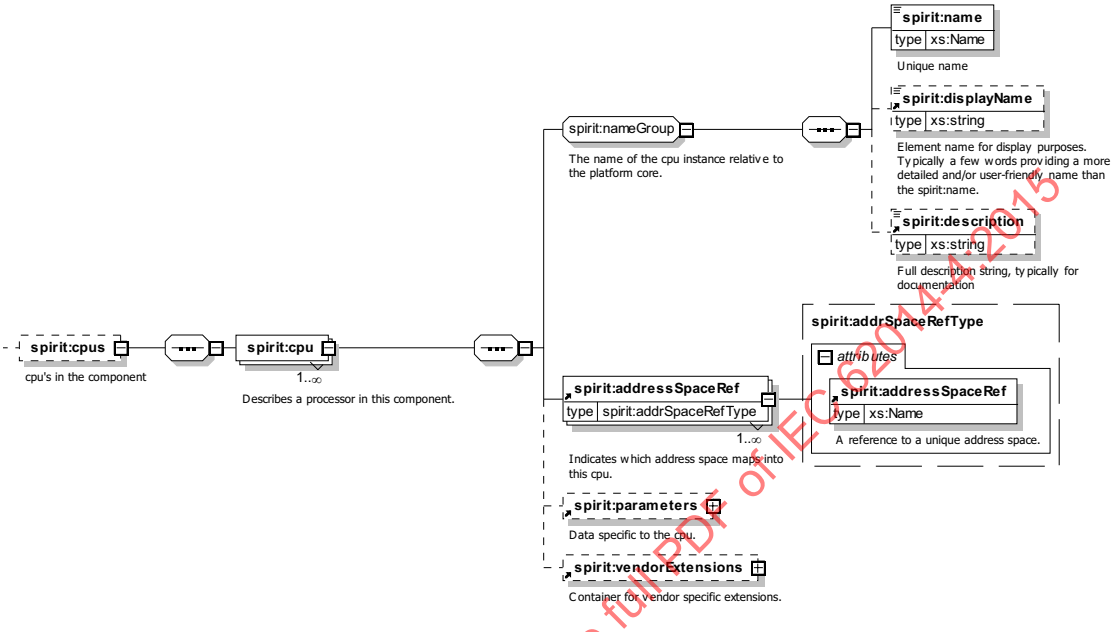
The following example shows the definition of a white box path to the status register bits in a component.

```
<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>
```

6.17 CPUs

6.17.1 Schema

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



6.17.2 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 departs.

a) ***nameGroup*** group is defined in C.1. The **name** element shall be unique within the containing **component** element.

b) **addressSpaceRef** (mandatory) 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 defines the range and width for transaction on this interface. See 6.7.1.

c) **parameters** (optional) specifies any **cpu**-type parameters. See C.11.

d) **vendorExtensions** (optional) adds any extra vendor-specific data related to the **cpu**. See C.10.

6.17.3 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>
```

7. Design descriptions

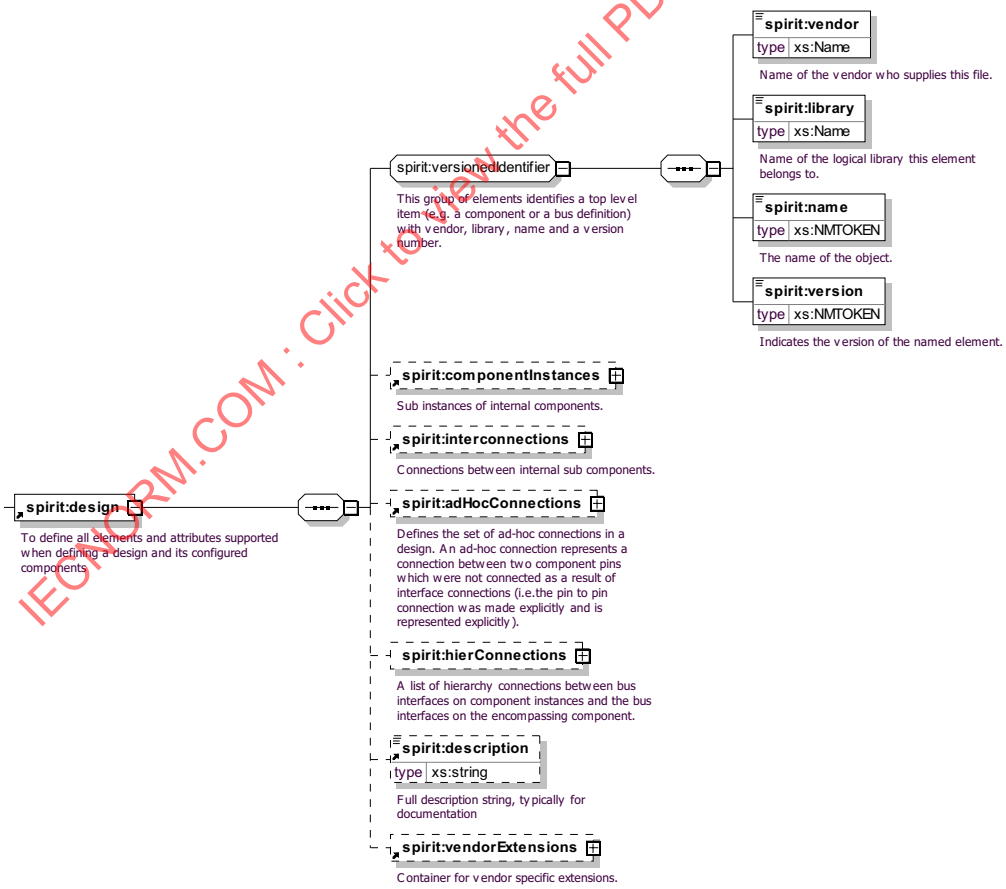
7.1 Design

An IP-XACT *design* is the central placeholder for the assembly of component objects meta-data. A design describes 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 description is analogous to a schematic of components.

While a design description, with referenced components and interconnections, 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 (referred to as a *hierarchical component*) is used. This *component description* contains a view with a reference to the design description. Together, the component and referenced design description form a complete single-level hierarchical description. From this point, it is simple to create additional hierarchical descriptions by including hierarchical component description in design descriptions.

7.1.1 Schema

The following schema details the information contained in the **design** element, which is one of the seven top-level elements of the schema.



7.1.2 Description

The **design** element describes 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 four subelements for a top level IP-XACT element. See C.6.

b) **componentInstances** (optional) contains the list of components that are instantiated (referenced) inside the design (see 7.2).

c) **interconnections** (optional) contains the list of connections between bus interfaces of components listed inside the design (see 7.3).

d) **adHocConnections** (optional) contains a list of connections between component ports listed inside this design (see 7.5).

e) **hierConnections** (optional) contains a list of connections between a component instance's bus interface and a bus interface inside the encompassing component (see 7.6). See also: 6.11.2.

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. See C.10.

See also: SCR 1.9.

7.1.3 Example

The following example shows as sample design with three components.

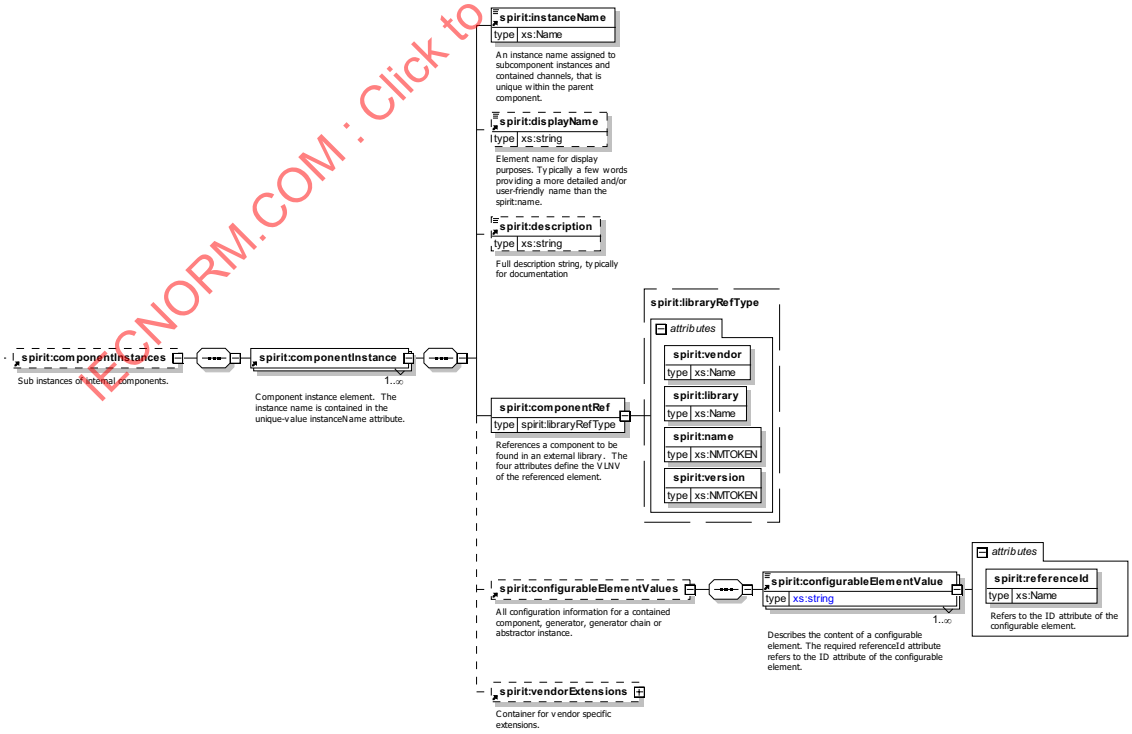
```
<spirit:design xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/
   SPIRIT/1.5" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5
   http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5/index.xsd">
   <spirit:vendor>spiritconsortium.org</spirit:vendor>
   <spirit:library>work</spirit:library>
   <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>master-channel-slave</spirit:description>
</spirit:design>
```

7.2 Design component instances

7.2.1 Schema

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



7.2.2 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** (mandatory) 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 **description** is of type ***string***.

d) **componentRef** (mandatory) is a reference to a component description (see 6.1) for this component instance. The componentRef element is of type ***libraryRefType*** (see C.7); it contains four attributes to specify a unique VLNV.

e) **configurableElementValues** (optional) specifies the configuration for a specific component instance by providing the value of a specific component parameter. The **configurableElements-Values** is an unbounded list of **configurableElementValue** elements.

   1) **configurableElementValue** (mandatory) is an unbounded list that specifies the value to apply to a configurable element; in this instance, it is pointed to by the **referenceId** attribute. The **configurableElementValue** is of type ***string***.

   2) The contained **referenceId** (mandatory) attribute 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 C.10.

See also: SCR 1.8 and SCR 5.14.

7.2.3 Example

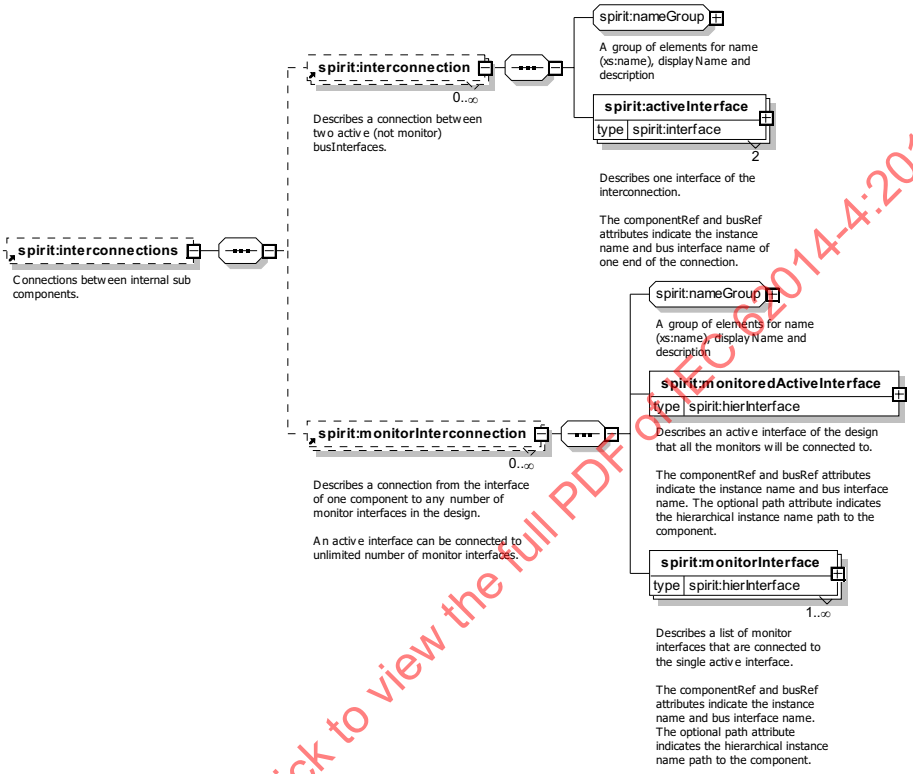
The following example shows two component instances of a design. The first one, `i_timers`, has a configurable element attached to it while the second one, `i_irqctrl`, is not configurable. The configurable element with the **id** equal to `TPRESC` has its value set to `22`.

```
<spirit:componentInstances>
   <spirit:componentInstance>
      <spirit:instanceName>i_timers</spirit:instanceName>
      <spirit:componentRef spirit:vendor="spiritconsortium.org"
         spirit:library="Leon2" spirit:name="timers"
spirit:version="1.5"/>
      <spirit:configurableElementValues>
         <spirit:configurableElementValue spirit:referenceId="TPRESC">22
         </spirit:configurableElementValue>
      </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="1.5"/>
   </spirit:componentInstance>
</spirit:componentInstances>
```

7.3 Design interconnections

7.3.1 Schema

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



7.3.2 Description

The **interconnections** element contains an unbounded list of **interconnection** and **monitorInterconnection** elements. For further description on interface connections, see 6.3.4.

a) **interconnection** (optional) 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 is defined in C.1. The **name** elements shall be unique within the containing **interconnections** element.

   2) **activeInterface** (mandatory) 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 7.4.

b) **monitorInterconnection** (optional) specifies the connection between a monitored active interface on a component and a list of monitor interfaces on component instances.

   1) ***nameGroup*** group is defined in C.1. The **name** elements shall be unique within the containing **interconnections** element.

   2) **monitoredActiveInterface** (mandatory) specifies the component bus interface to monitor. Only one monitored active interface is allowed. The **monitoredActiveInterface** element is of type ***hierInterface***, see 7.4.

3) **monitorInterface** (mandatory) specifies the component bus interface that will do the monitoring. There may be one or more **monitorInterface** elements specified. The **monitorInterface** element is of type ***hierInterface***, see 7.4.

See also: SCR 6.9, SCR 6.10, and SCR 6.14 and the SCRs in Table B.2 and Table B.4.

7.3.3 Example

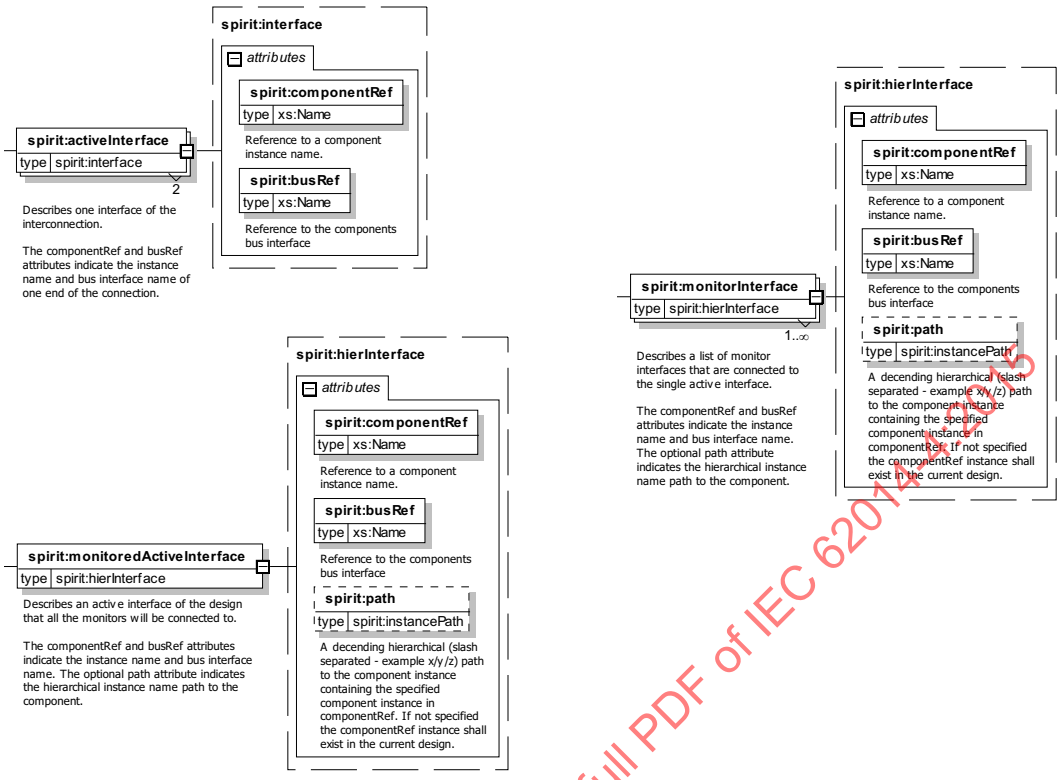
The following example shows two interconnections between three components: the interconnection `interco1` connects the interface `ambaAPB` on `i_timers` to the interface `MirroredSlave0` on `i_apbbus` while `interco2` connects the interface `ambaAPB` on `i_irqctrl` to the interface `MirroredSlave1` on `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>
```

7.4 Active, monitored, and monitor interfaces

7.4.1 Schema

The following schema details the information contained in the **activeInterface** element, the **monitoredActiveInterface** element, and the **monitorInterface** elements, which may appear as an element inside the **interconnection** or **monitorInterconnection** element within the **interconnections** element.



7.4.2 Description

The **activeInterface**, **monitoredInterface**, or **monitorInterface** element specifies the bus interface of a design component instance that is part of an interconnection or a monitor interconnection. They all have the following attributes.

a) **componentRef** (mandatory) references the instance name of a component present in the design if the path attribute is not present. This component instance name needs to exist in the specified design. The **componentRef** attribute is of type ***Name***. See 6.1.

b) **busRef** (mandatory) references one of the component bus interfaces. This specific bus interface needs to exist on the specified component instance. The **busRef** attribute is of type ***Name***. See 6.5.

The **monitoredActiveInterface** and **monitorInterface** elements have the following attribute.

path (optional) defines the hierarchical path of instance names to the design that contains the component instance specified in the **componentRef** attribute. The path is a slash (/) separated list of instance names. If the **path** attribute is not present, the component referenced by **componentRef** needs to exist in the current design. The **path** attribute is of type ***instancePath***. See D.5.

See also: SCR 2.1, SCR 2.16, SCR 4.1, and SCR 4.2.

7.4.3 Example

The following example shows a monitored interface referring to the `ambaAPB` bus interface on the component instance `i_timers` in the design within the component with instance name `apbsubsys/group1` and a monitor interface referring to the `ambaAPBMonitor` bus interface on the monitor instance `i_monitor` in the design within the component with instance name `umon`.

```
<spirit:monitoredInterface spirit:path="apbsubsys/group1"
   spirit:componentRef="i_timers" spirit:busRef="ambaAPB"/>
<spirit:monitorInterface spirit:path="umon" spirit:componentRef="i_monitor"
   spirit:busRef="ambaAPBMonitor"/>
```

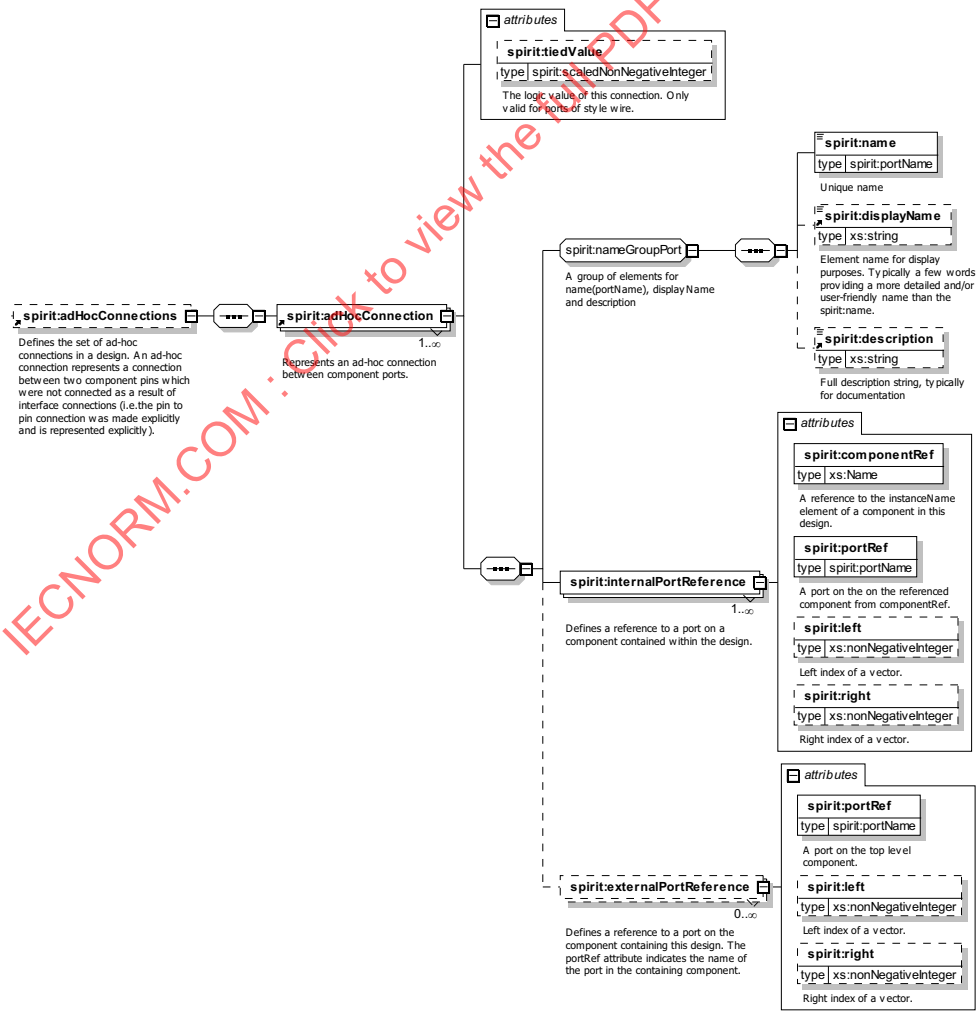
7.5 Design ad hoc connections

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** that 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.

7.5.1 Schema

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



7.5.2 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 is defined in C.1. The **name** elements shall be unique within the containing **adHocConnections** element.

b) **internalPortReference** (mandatory) 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***. See 6.1.

   2) **portRef** (mandatory) references the port name on the specific component instance. The **portRef** attribute is of type ***Name***. See 6.11.3.

   3) **left** and **right** (optional) specify a portion of the port range. The **left** and **right** attributes are of type ***nonNegativeInteger***.

c) **externalPortReference** (optional) 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***. See 6.11.3.

   2) **left** and **right** (optional) specify a portion of the port range. The **left** and **right** attribute is of type ***nonNegativeInteger***.

See also: SCR 6.14.

7.5.3 Example

The following example shows two ad hoc connections. The first one, `d1e1074`, connects port `irlin` on component instance `i_irqctrl` and port `irqvec` on component instance `i_leon2Proc`. The second one, `i_leon2Proc_mresult`, connects 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>
```

7.5.4 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 bits of the ports are connected from left to right. In the **internalPortReference** element, **left** and **right** define the actual bits to connect.

See also: SCR 6.9 and SCR 6.27.

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_logic` port.

7.5.5 Ad hoc transactional connection

For ad hoc transactional connections, IP-XACT requires:

— The style of each port be the same style (i.e., **transactional**).

— If defined, the **transTypeDef/typeName** name of each port are the same (e.g., `sc_tlm_port`).

— The **service/serviceTypeDef/typeName**s match.

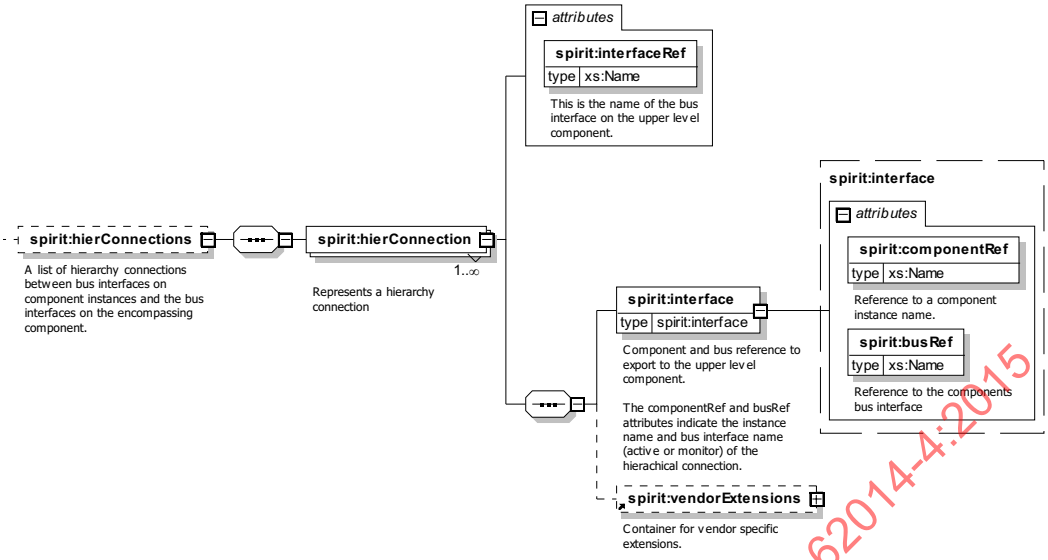
Also, two ports with a **requires** initiative can be connected. This means they would both connect to a mediated link (e.g., a wire, buffer, FIFO, or any complex link) in a top SystemC or SystemVerilog netlist. This mediated link provides the protocol interfaces required by each port. The name, type, and parameters of this mediated link are not defined by IP-XACT, but could be given as input to a netlister generator.

See also: SCR 6.10.

7.6 Design hierarchical connections

7.6.1 Schema

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



7.6.2 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 (see 6.5.1). The **interfaceRef** attribute is of type ***Name***. The name of the ports and the mapping to this interface are defined in the referencing hierarchical component. The **hierConnection** element contains the following elements and attributes.

a) **interface** (mandatory) specifies the component instance bus interface for connection to the encompassing component; only one **interface** is allowed. The **interface** element may reference an active interface or a monitor interface. The **interface** element is of type ***interface***, see 7.4.

b) **vendorExtensions** (optional) adds any extra vendor-specific data related to the hierarchical interface connection. See C.10.

See also: SCRs in Table B.10 and Table B.11.

7.6.3 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>
```

8. Abstractor descriptions

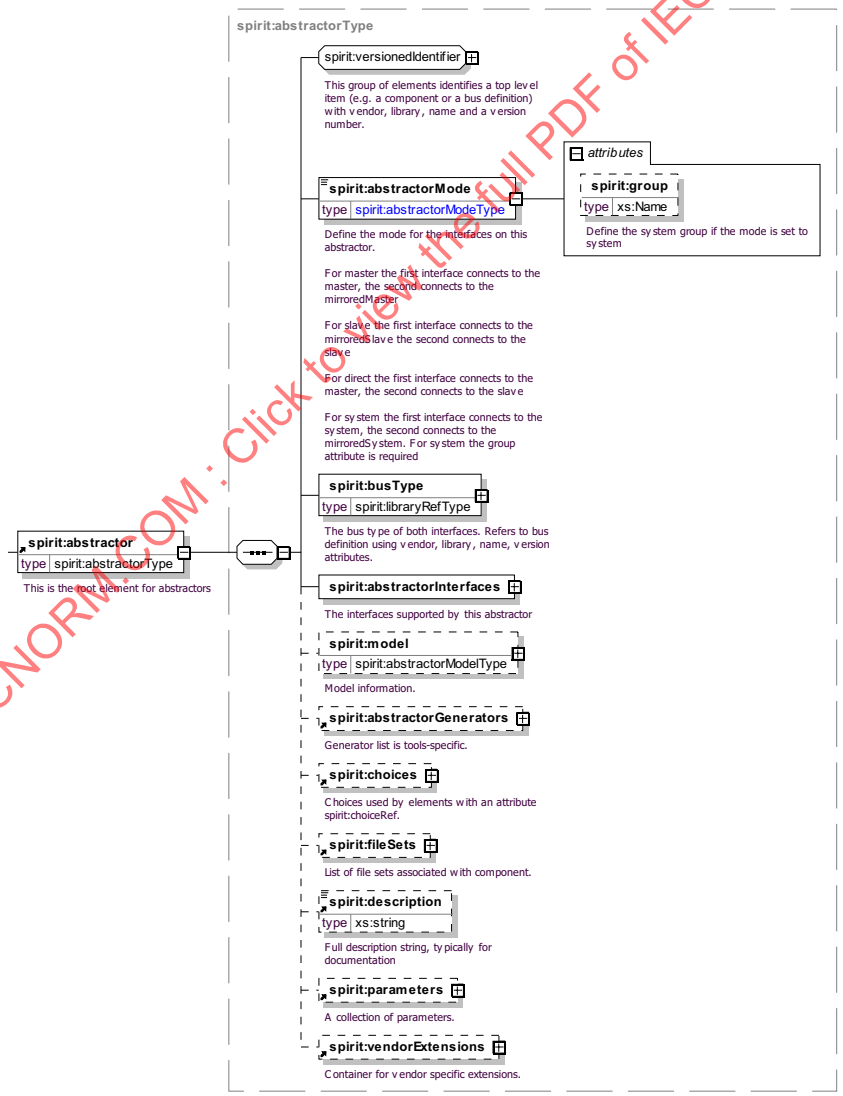
Designs that incorporate IP models using different interface modeling styles (e.g., TLM and RTL modeling styles) may contain interconnections between such component interfaces using different abstractions of the same bus type. An IP-XACT description may describe how such interconnections are to be made using a special-purpose object called an *abstractor*. An abstractor is used to connect between two different abstractions of the same bus type (e.g., an APB_RTL and an APB_TLM). An abstractor shall only contain two interfaces, which shall be of the same bus definition and different abstraction definitions.

Unlike a component, an abstractor is not referenced from a design description, but instead is referenced from a design configuration description. See Clause 10.

8.1 Abstractor

8.1.1 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.



8.1.2 Description

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

a) ***versionedIdentifier*** group provides a unique identifier, made up of four subelements for a top-level IP-XACT element. See C.6.

b) **abstractorMode** (mandatory) determines the mode of the two interfaces contained in **abstractorInterfaces**. The abstractor can be inserted in a connection between two instances or between an instance and an exported interface. The **abstractorMode** element can take one of the following four values.

1) **master** specifies for

i) master to mirrored-master connection—the first interface connects to the master interface, the second connects to the mirrored-master interface;

ii) exported master connection—the first interface connects to the master interface, the second connects to the exported interface;

iii) exported mirrored-master connection—the first interface connects to the exported interface, the second connects to the mirrored-master interface.

2) **slave** specifies for

i) mirrored-slave to slave connection—the first interface connects to the mirrored-slave interface, the second connects to the slave interface;

ii) exported slave connection—the first interface connects to the exported interface, the second connects to the slave interface;

iii) exported mirrored-slave connection—the first interface connects to the mirrored-slave interface, the second connects to the exported interface.

3) **direct** specifies the first interface connects to the master interface, the second connects to the slave interface. This option is not allowed for an exported interface.

4) **system** specifies for

i) system to mirrored-system connection—the first interface connects to the system interface, the second connects to the mirrored-system interface;

ii) exported system connection—the first interface connects to the system interface, the second connects to the exported interface;

iii) exported mirrored-system connection—the first interface connects to the exported interface, the second connects to the mirrored-system interface.

The **group** (mandatory, when **abstractorMode="system"**) attribute defines the name of the group to which this system interface belongs. This attribute is of type ***Name***, which indicates the value of this group shall be unique inside the **abstractor** element. 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.

c) **busType** (mandatory) specifies the bus definition this bus interface references. A bus definition (see 5.2) describes the high-level attributes of a bus description. The **busType** element is of type ***libraryRefType*** (see C.7); it contains four attributes to specify the referenced VLNV.

d) **abstractorInterfaces** (mandatory) are interfaces having the same bus type, but differing abstraction types. See 8.2.

e) **model** (optional) specifies all the different views, ports, and model configuration parameters of the abstractor. See 8.3.

f) **abstractorGenerators** (optional) specifies a list of generator programs attached to this abstractor. See 8.7.

g) **choices** (optional) specifies multiple enumerated lists, which are referenced by other sections of this abstractor description. See 6.14.

h) **fileSets** (optional) specifies groups of files and possibly their function for reference by other sections of this abstractor description. See 6.13.

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

j) **parameters** (optional) describes any **parameter** that can be used to configure or hold information related to this abstractor. See C.11.

k) **vendorExtensions** (optional) contains any extra vendor-specific data related to the abstractor. See C.10.

See also: SCR 1.9, SCR 1.10, and SCR 3.16.

8.1.3 Example

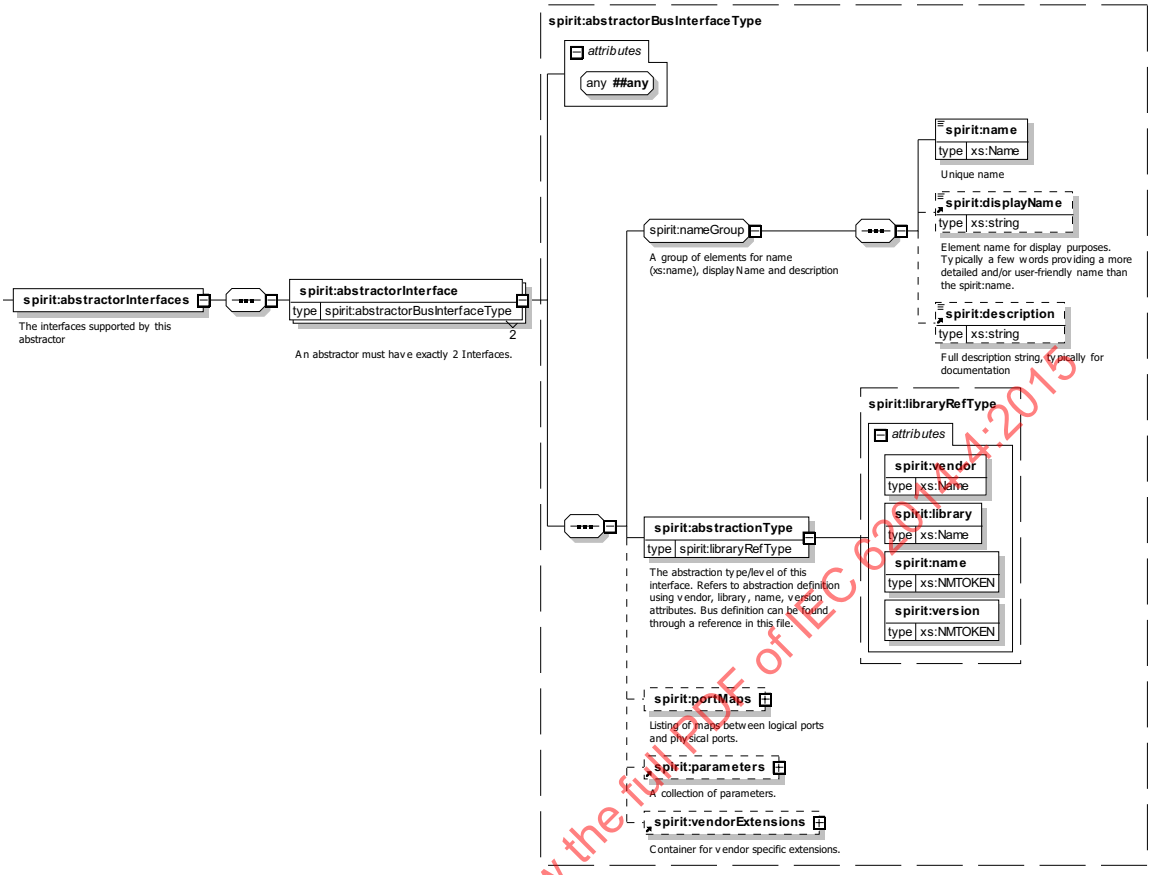
The following example shows a simple slave abstractor having `AHB UT` and `AHB LT` interfaces.

```
<spirit:abstractor>
   <spirit:vendor>spiritconsortium.org</spirit:vendor>
   <spirit:library>Leon2</spirit:library>
   <spirit:name>pv2rtl</spirit:name>
   <spirit:version>1.5</spirit:version>
   <spirit:abstractorMode>slave</spirit:abstractorMode>
   <spirit:busType spirit:vendor="amba.com" spirit:library="AMBA2"
   spirit:name="AHB" spirit:version="r2p0_5"/>
   <spirit:abstractorInterfaces>
      <spirit:abstractorInterface>
         <spirit:name>UTinterface</spirit:name>
         <spirit:abstractionType
            spirit:vendor="spiritconsortium.org"
            spirit:library="Leon2"
            spirit:name="AHB_UT"
            spirit:version="1.0"/>
      </spirit:abstractorInterface>
      <spirit:abstractorInterface>
         <spirit:name>LTinterface</spirit:name>
         <spirit:abstractionType
            spirit:vendor="spiritconsortium.org"
            spirit:library="Leon2"
            spirit:name="AHB_LT"
            spirit:version="1.0"/>
      </spirit:abstractorInterface>
   </spirit:abstractorInterfaces>
</spirit:abstractor>
```

8.2 Abstractor interfaces

8.2.1 Schema

The following schema defines the information contained in the **abstractorInterfaces** element, which appears within an **abstractor** description.



8.2.2 Description

The **abstractorInterfaces** element contains a list of two **abstractorInterface** elements. Each **abstractorInterface** element defines properties of this specific interface in an abstractor. The **abstractorInterface** element also allows for vendor attributes to be applied. Each **abstractorInterface** contains the following elements:

a) ***nameGroup*** group is defined in C.1. The **name** elements shall be unique within the containing **abstractor** element.

b) **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 5.3). The **abstractionType** element is of type ***libraryRefType*** (see C.7); it contains four attributes to specify the referenced VLNV.

c) **portMaps** (optional) describes the mapping between the abstraction definition's logical ports and the abstractor's physical ports. See 6.5.6.

d) **parameters** (optional) specifies any parameter data value(s) for this bus interface. See C.11.

e) **vendorExtensions** (optional) holds any vendor-specific data from other namespaces, which is applicable to this bus interface. See C.10.

8.2.3 Example

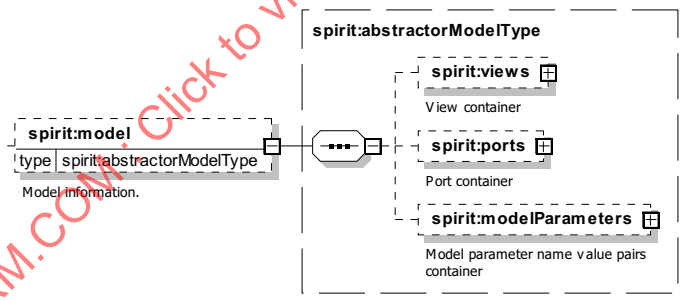
This example shows an **abstractorInterface** of type `AHB_PV`, which includes a single `portMap` between the logical port `PV_TRANS` and the abstractor physical port `ahb_slave_port`.

```
<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:portMaps>
        <spirit:portMap>
            <spirit:logicalPort>
                <spirit:name>PV_TRANS</spirit:name>
            </spirit:logicalPort>
            <spirit:physicalPort>
                <spirit:name>ahb_slave_port</spirit:name>
            </spirit:physicalPort>
        </spirit:portMap>
    </spirit:portMaps>
</spirit:abstractorInterface>
```

8.3 Abstractor models

8.3.1 Schema

The following schema defines the information contained in the abstractor **model** element, which may appear within an **abstractor** description.



8.3.2 Description

The **model** element describes the views, ports, and model related parameters of an abstractor. 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 software 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 8.4.

b) **ports** (optional) contains the list of ports for this object. A ports is an external connection from the object. An object may only have one set of ports that shall be valid for all views. See 8.5.

c) **modelParameters** (optional) contains a list of parameters that are needed to configure a model implementation. The same set of model parameters shall be valid for all views. See 6.11.20.

8.3.3 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>
```

8.4 Abstractor views

8.4.1 Schema

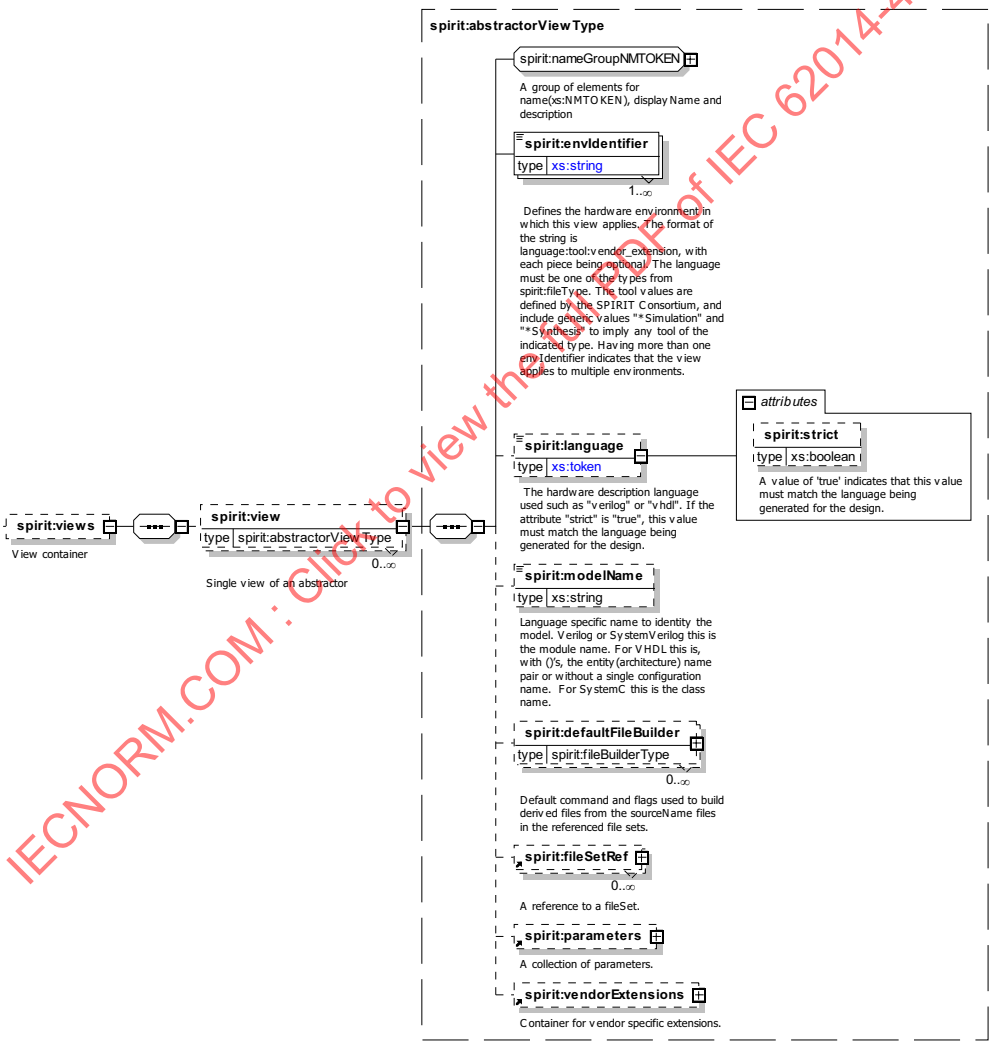
The following schema defines the information contained in the **views** element, which appears within the **model** element of an **abstractor** description.

This schema is almost identical to the **component/views/view** element (see 6.11.2), except:

— Abstractors have no **hierarchyRef** elements.

— Abstractors have no **constraintSetRef** elements.

— Abstractors have no **whiteboxElementRefs** elements.



8.4.2 Description

A **views** element describes an unbounded set of **view** elements. Each **view** element specifies a representation level of an abstractor. It contains the following elements.

a) ***nameGroupNMToken*** group is defined in C.4. The **name** elements shall be unique within the containing **views** element.

b) **envIdentifier** (mandatory) 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 colons [**:**] in the format of *Language***:***Tool***:***VendorSpecific*. The regular expression that is used to check the string is [A-Za-z0-9_+*\.]*:[A-Za-z0-9_+*\.]*:[A-Za-z0-9_+*\.]* The sections are:

1) *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.

2) *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 IP-XACT standard tool names for **envIdentifier** [B14].

3) *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) **language** (optional) specifies the HDL used for a specific view, e.g., `verilog`, `vhdl`, or `SystemC`. The **language** element 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). 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**.

d) **modelName** (optional) is a language-specific identifier of the model. For Verilog or SystemVerilog, this is the module name. For VHDL, this is, with `()`'s, the entity (architecture) name pair or, without `()`'s, a configuration name. For SystemC, this is the `sc_module` class name. The **modelName** element is of type ***string***.

e) **defaultFileBuilder** (optional) is an unbounded list of default file builder options for the **fileSets** referenced in this **view**. See 6.13.5.

f) **fileSetRef** (optional) is an unbounded list of references to a **fileSet name** within the containing document or another document referenced by the **VLNV**. See C.8.

g) **parameters** (optional) details any additional parameters that describe the **view** for generator usage. See C.11.

h) **vendorExtensions** (optional) adds any extra vendor-specific data related to the view. See C.10.

8.4.3 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.

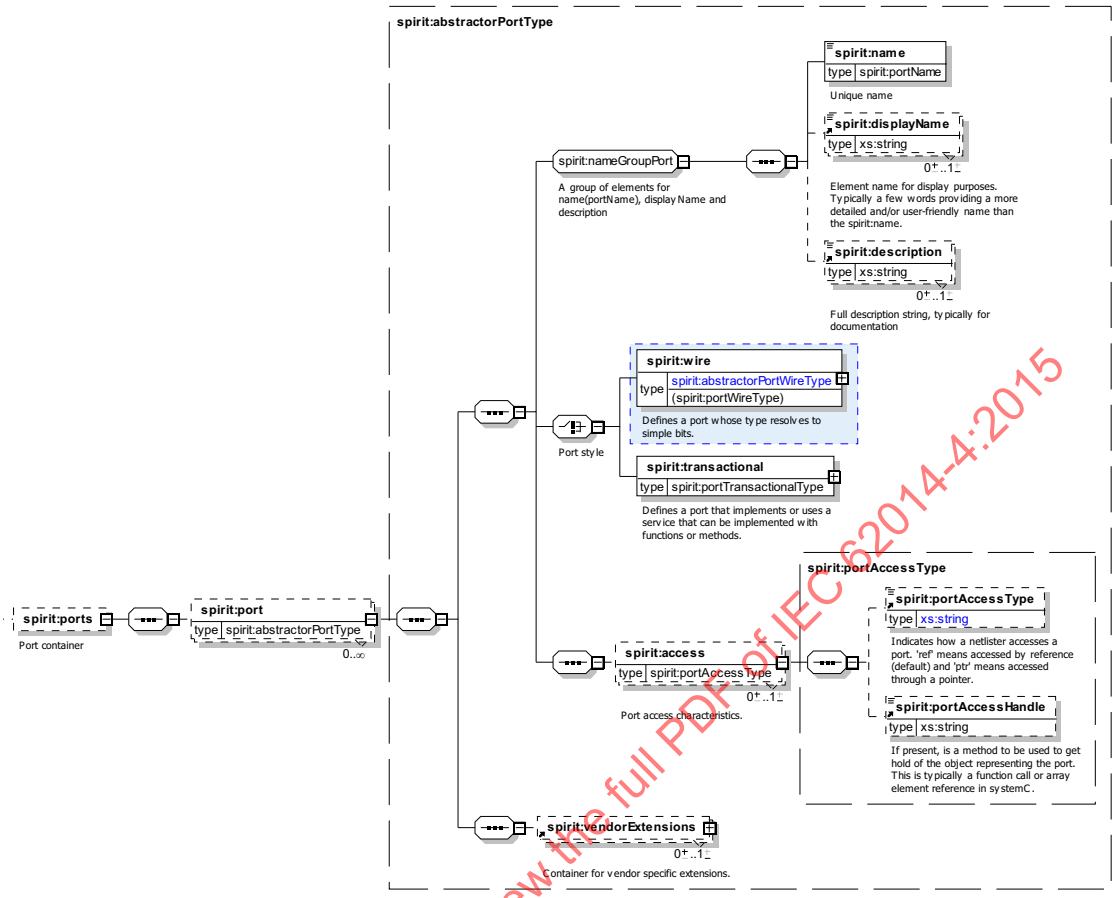
```
<spirit:views>
   <spirit:view>
      <spirit:name>systemCView</spirit:name>
      <spirit:envIdentifier>:*Simulation:</spirit:envIdentifier>
      <spirit:language>systemc2.1</spirit:language>
      <spirit:modelName>pv2pvt</spirit:modelName>
      <spirit:fileSetRef>
         <spirit:localName>scFileSetRef</spirit:localName>
      </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>
         <spirit:localName>svFileSetRef</spirit:localName>
      </spirit:fileSetRef>
   </spirit:view>
</spirit:views>
```

8.5 Abstractor ports

8.5.1 Schema

An abstractor's **ports** are almost identical to a component's **ports**; the abstractor **transactional** ports are exactly the same as the component **transactional** ports. The access methods are the same for an abstractor or component port. 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.

The following schema defines the information contained in the **ports** element, which may appear within an **abstractor**.



8.5.2 Description

The **ports** element defines an unbounded list of **port** elements. Each **port** element describe a single external port on the abstractor.

a) ***nameGroupPort*** group is defined in C.4. The **name** elements shall be unique within the containing **ports** element.

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. A wire port in an abstractor contains most of the same elements and attributes as a wire port in a component, except for the **contraintSet** element. See 8.6.

2) **transactional** (mandatory) defines all other style ports, typically used for TLM. A transactional port in an abstractor contains all the same elements and attributes as a transactional port in a component. See 6.11.16.

c) **access** (optional) defines the access for a port.

1) **portAccessType** (optional) indicates to a netlister how to access the port. The **portAccessType** shall have one of two possible values **ref** or **ptr**. If **ref** (the default), a netlister should access the port directly, and if **ptr**, it should access the port with a pointer.

2) **portAccessHandle** (optional) indicates to a netlister the method to be used to access the object representing the port. This is typically a function call or array element reference in IEEE Std 1666-2005 [B4] (SystemC). The **portAccessHandle** is of type ***string***.

d) **vendorExtensions** (optional) adds any extra vendor-specific data related to the port. See C.10.

8.5.3 Example

The following example shows a simple address port with a transactional interface.

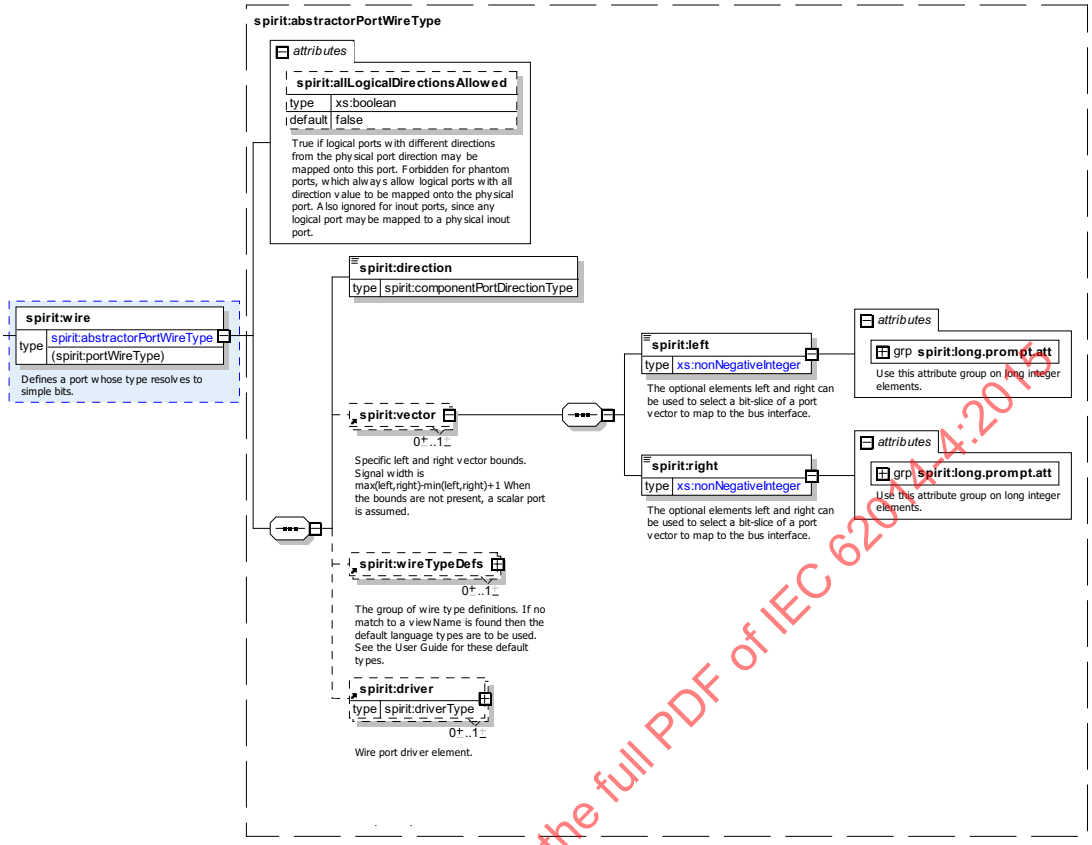
```
<spirit:ports>
   <spirit:port>
      <spirit:name>paddr</spirit:name>
      <spirit:transactional>
         <spirit:service>
          <spirit:initiative>provides</spirit:initiative>
            <spirit:serviceTypeDefs>
            <spirit:serviceTypeDef>
            <spirit:typeName>trans_if</spirit:typeName>
            <spirit:parameters>
            <spirit:parameter name=”addr” resolve=”user”>ADDR
            </spirit:parameter>
            </spirit:parameters>
            </spirit:serviceTypeDef>
            </spirit:serviceTypeDefs>
            </spirit:service>
      </spirit:transactional>
   </spirit:port>
</spirit:ports>
```

8.6 Abstractor wire ports

8.6.1 Schema

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.

The following schema element defines the information contained in the **wire** element, which appears within an abstractor **port**.



8.6.2 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 a hardware 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 whether the port may be mapped to a port in an **abstractionDefinition** with a different direction. The default value is **false**. The **allLogical-DirectionsAllowed** attribute is of type ***boolean***. See 5.3.

b) **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 that only exists on the IP-XACT component, but not on the implementation referenced from the view.

c) **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 C.12.

1) 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.
2) 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*.

d) **wireTypeDefs** (optional) describes the ports type as defined by the implementation, see 6.11.5.
e) **driver** (optional) defines a driver that 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 6.11.6.

See also: SCR 6.5, SCR 6.6, SCR 6.7, and SCR 6.12.

8.6.3 Example

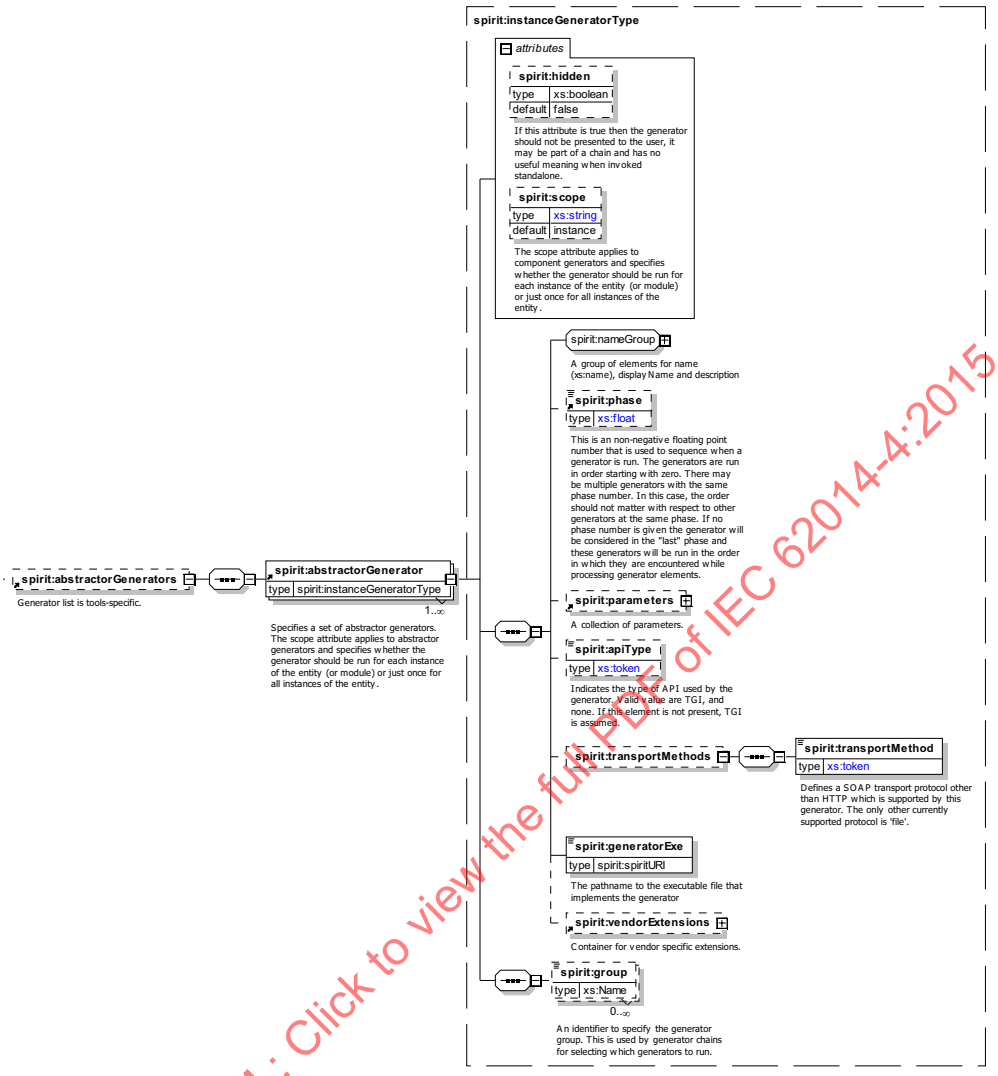
The following example shows a simple address port of 32 bits.

```
<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>
```

8.7 Abstractor generators

8.7.1 Schema

The following schema defines the information contained in the **abstractorGenerators** element, which may appear within an **abstractor** object.



8.7.2 Description

The **abstractorGenerators** element contains an unbounded list of **abstractorGenerator** elements. Each **abstractorGenerator** element defines a generator that is assigned and may be run on this abstractor. The **abstractorGenerator** has exactly the same schema definition as a **componentGenerator**. See 6.12.

8.7.3 Example

The following example shows a document generator attached to an abstractor. This generator is a Tcl script that can be executed as `tclsh ../bin/absDocGen.tcl -url` *`file`* (and `useDefaultValues` is `true`). Here, the parameter is a configurable parameter named `useDefaultValues`, which can be configured by the user. 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>
```

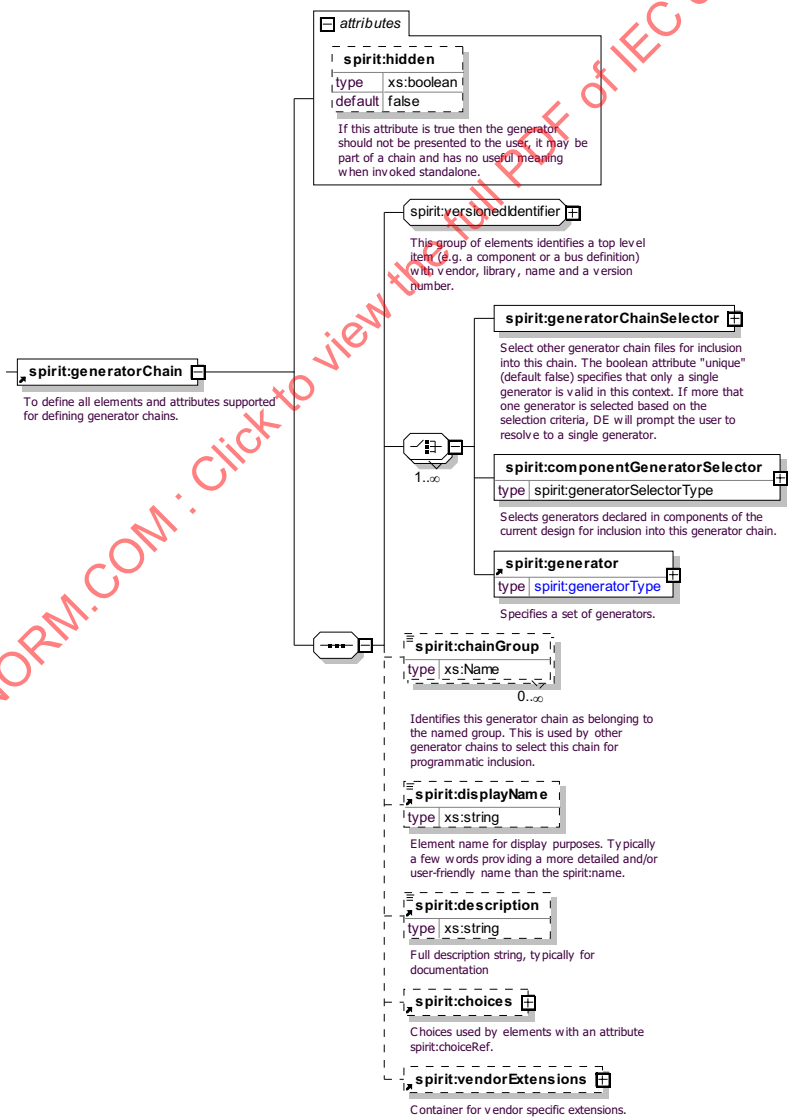
9. Generator chain descriptions

9.1 generatorChain

In IP-XACT, a design flow can be represented as a generator chain. A generator chain is an ordered sequence of named tasks. Each named task can be represented as a single generator or as another 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 generator chain description.

9.1.1 Schema

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



9.1.2 Description

The **generatorChain** element describes a single generator chain. The **generatorChain** element contains a **hidden** (optional) attribute that, when **true**, indicates this generator chain is not presented to the user of a DE. This may be the case if the chain is part of another chain and has no useful meaning when invoked as stand-alone. The default is **false**. The **hidden** attribute is of type ***boolean***. The **generatorChain** element contains the following elements.

a) ***versionedIdentifier*** group provides a unique identifier; it consists of four subelements for a top-level IP-XACT element. See C.6.

b) One or more of the following three elements.

   1) **generatorChainSelector** (optional) is a selection criteria for selecting one or more **generatorChains** or a reference to another **generatorChain** (see 9.2).

   2) **componentGeneratorSelector** (optional) is a selection criteria for selecting one or more component generators (see 9.3).

   3) **generator** (optional) defines the generator (see 9.4).

c) **chainGroup** (optional) is an unbounded list of names to which this chain belongs. The group names are referenced in the **generatorChainSelector** element and can be used to organize the inclusion of generators. The **chainGroup** element is of type ***Name***.

d) **displayName** (optional) allows a short descriptive text to be associated with the generator chain. The **displayName** element is of type ***string***.

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

f) **choices** (optional) specifies multiple enumerated lists, which are referenced by other sections of this generator chain description. See 6.14.

g) **vendorExtensions** (optional) contains any extra vendor-specific data related to the **generatorChain**. See C.10.

See also: SCR 1.9.

9.1.3 Example

The following example defines a generator chain with a group name of `MY_HW_SW_COMPILATION_CHAIN`, which is intended to specify a sequence of four simulation tasks (e.g., `INIT`, `CONFIG`, `BUILD`, and `COMPILE`) for both hardware and software compilation.

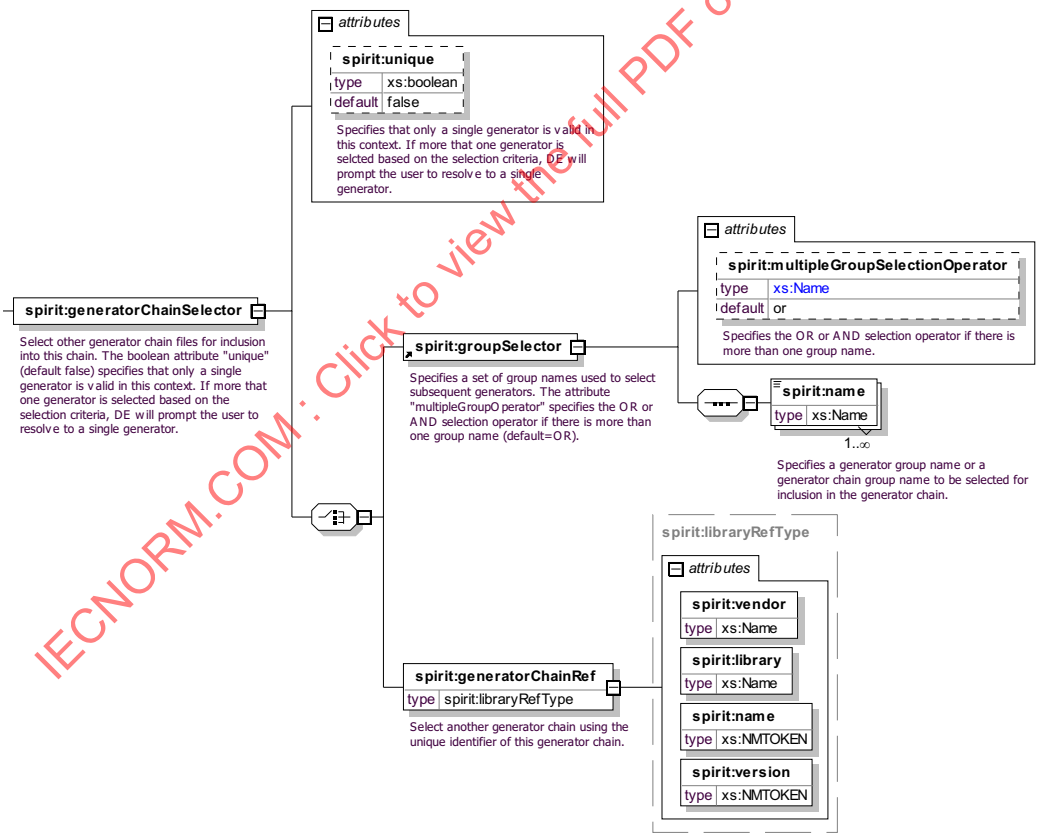
```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:generatorChain
    xmlns:xs=http://www.w3.org/2001/XMLSchema
    xmlns:spirit=http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5
    xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5
    http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5/index.xsd">
    <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>INIT</spirit:name>
        </spirit:groupSelector>
    </spirit:generatorChainSelector>
    <spirit:generatorChainSelector>
        <spirit:groupSelector>
            <spirit:name>CONFIG</spirit:name>
```

```
        </spirit:groupSelector>
    </spirit:generatorChainSelector>
    <spirit:generatorChainSelector>
        <spirit:groupSelector>
            <spirit:name>BUILD</spirit:name>
        </spirit:groupSelector>
    </spirit:generatorChainSelector>
    <spirit:generatorChainSelector>
        <spirit:groupSelector>
            <spirit:name>COMPILE</spirit:name>
        </spirit:groupSelector>
    </spirit:generatorChainSelector>
    <spirit:chainGroup>MY_HW_SW_COMPILATION_CHAIN</spirit:chainGroup>
</spirit:generatorChain>
```

9.2 generatorChainSelector

9.2.1 Schema

The following schema defines the information contained in the **generatorChainSelector** element, which may appear within a **generatorChain**.



9.2.2 Description

The **generatorChainSelector** element defines which generator(s) to invoke based on a selection criteria. The **generatorChainSelector** element contains a **unique** (optional) attribute that, when **true**, indicates the generatorChainSelector shall resolve to a single generator. If more than one generator is selected, the DE

shall resolve the selection to a single generator. The **unique** attribute default is **false** and is of type ***boolean***. The **generatorChainSelector** element can specify the selection criteria in one of two ways: as a selection based on the **chainGroup** names via the **groupSelector** element or as a direct VLNV reference via the **generatorChainRef** element. The **generatorChainSelector** element shall contain one of the **groupSelector** or **generatorChainRef** elements.

a) **groupSelector** (mandatory) is a container for an unbounded list of chain group **name** elements.

   1) When more than one **name** element is specified, the **multipleGroupSelectorOperator** (optional) attribute can specify if the selection applies when *one* of the generator group names matches (**multipleGroupSelectorOperator** equals **or**) or *all* the generator group names match (**multipleGroupSelectorOperator** equals **and**).

   2) **name** (mandatory) is an unbounded list of selection names. The names are compared to the **generatorChain/chainGroup** elements within all generator chains visible to the DE. The **name** element is of type ***Name***.

b) **generatorChainRef** (mandatory) specifies a reference to another generator chain description for inclusion in this generator chain. The **generatorChainRef** element is of type ***libraryRefType*** (see C.7); it contains four attributes to specify a unique VLNV.

See also: SCR 1.7.

9.2.3 Example

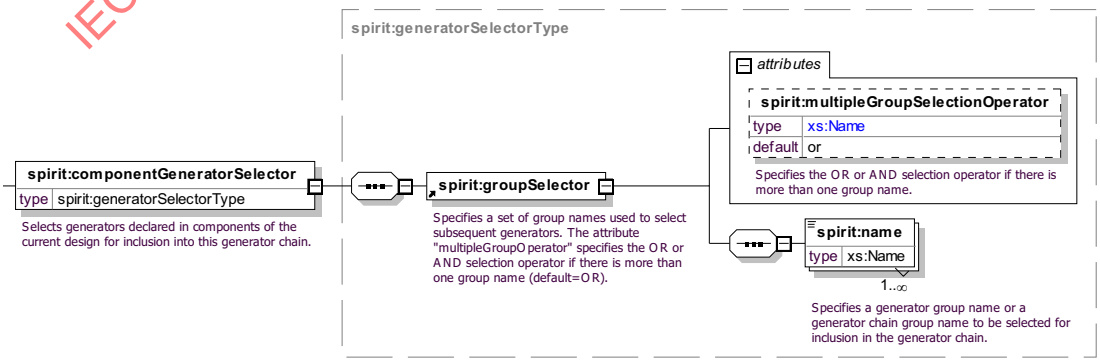
Assume three **generatorChain**s `X`, `Y`, and `Z` have been created with the **chainGroup** names `{A, B}`, `{A, C}`, and `{B, C}`, respectively. This example shows how a new **generatorChain** object can select `Y`.

```
<spirit:generatorChainSelector>
   <spirit:groupSelector spirit:multipleGroupSelectionOperation=”and“>
      <spirit:name>A</spirit:name>
      <spirit:name>C</spirit:name>
   </spirit:groupSelector>
</spirit:generatorChainSelector>
```

9.3 generatorChain component selector

9.3.1 Schema

The following schema defines the information contained in the **componentGeneratorSelector** element, which may appear within a **generatorChain**.



9.3.2 Description

Similar to the **generatorChainSelector**, **componentGeneratorSelector** selects a component generator or a list of component generators based on the assigned group name. The **componentGeneratorSelector** contains the **groupSelector** element.

groupSelector (mandatory) is a container for an unbounded list of chain group **name** elements.

1) When more than one **name** element is specified, the **multipleGroupSelectorOperator** (optional) attribute can specify if the selection applies when *one* of the generator group names matches (**multipleGroupSelectorOperator** equals **or**) or *all* the generator group names match (**multipleGroupSelectorOperator** equals **and**).
2) **name** (mandatory) is an unbounded list of selection names. The names are compared to the **componentGenerator**/**group** elements within all components in the current design. The **name** element is of type ***Name***.

9.3.3 Example

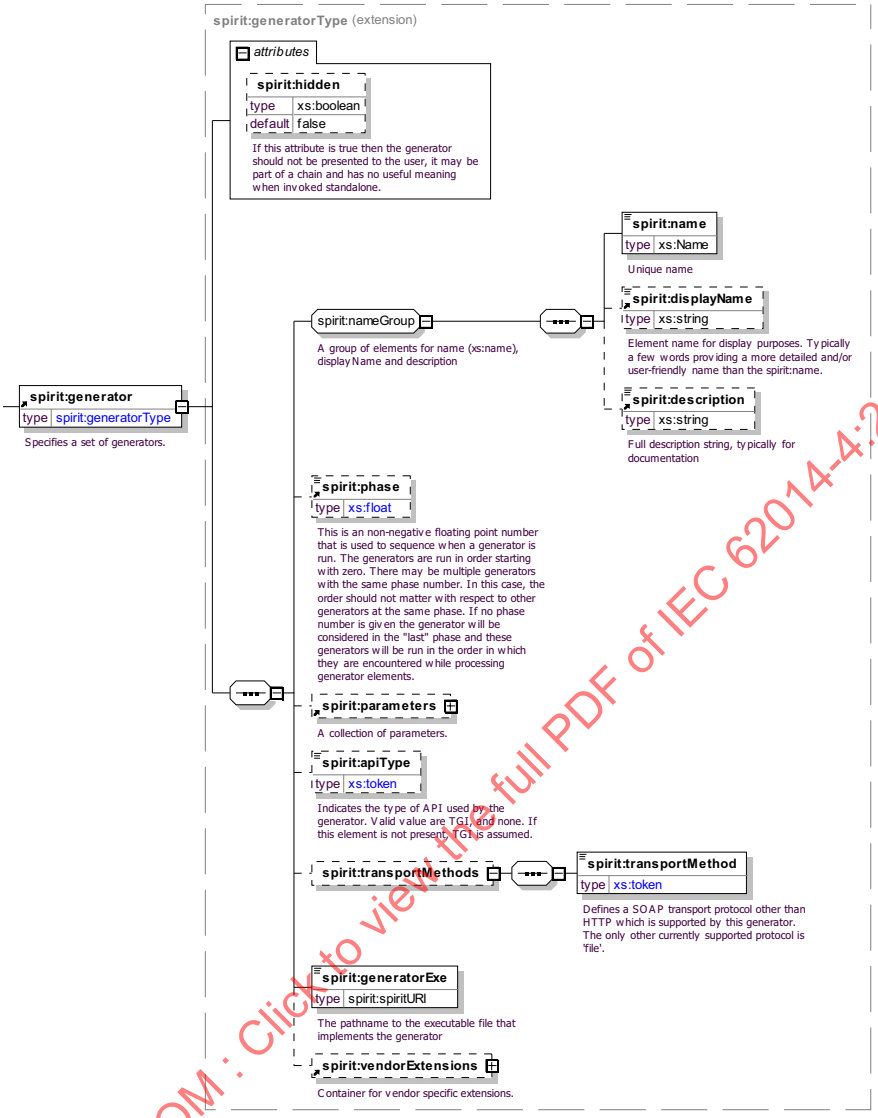
The following example shows a **generatorChain** selecting all the component generators whose **group** element matches the name `docGen`.

```
<spirit:componentGeneratorSelector>
   <spirit:groupSelector>
      <spirit:name>docGen</spirit:name>
   </spirit:groupSelector>
</spirit:componentGeneratorSelector>
```

9.4 generatorChain generator

9.4.1 Schema

The following schema defines the information contained in the **generator** element, which may appear within a **generatorChain**.



9.4.2 Description

The **generator** element defines a specific generator executable. The **generator** element contains a **hidden** attribute. The **hidden** (optional) attribute specifies, when **true**, this generator shall not be run as a stand-alone generator and is required to be run as part of a chain. This generator is not presented to the user. If **false** (the default), this generator may be run as a stand-alone generator or in a generator chain. The **hidden** attribute is of type ***boolean***. **generator** contains the following elements.

a) ***nameGroup*** group is defined in C.1.

b) **phase** (optional) determines the sequence in which a generators are run. Generators are run in order starting with zero (0). If two 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 any order after the last generator with a phase number. The **phase** element is of type ***float*** and shall also be a positive number.

c) **parameters** (optional) specifies any **generator** type parameters. See C.11.

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 DE 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 via 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 **generator**. See C.10.

9.4.3 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>
```

10. Design configuration descriptions

10.1 Design configuration

An IP-XACT *design configuration* is a placeholder for additional configuration information of a design or generator chain description. Design configuration information is useful when transporting designs between design environments and automating generator chain execution for a design, by storing information that would otherwise have to be re-entered by the designer.

The *design configuration description* contains the following configuration information:

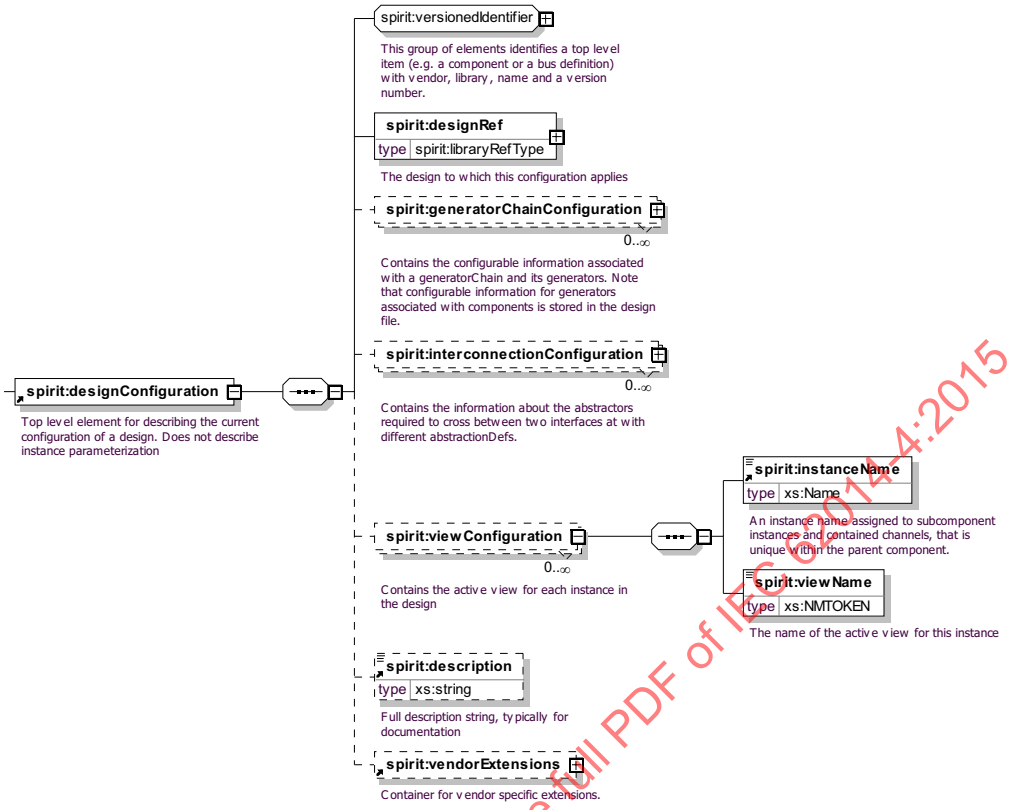
- Configurable information for parameters defined in generators within generator chains; this information is not referenced via the design description;
- Active view or current view selected for instances in the design description;
- Configuration information for interconnections between the same bus types with differing abstraction types (i.e., abstractor reference, parameter configuration, and view selection). See also Clause 8.

A design configuration applies to a single design, but a design may have multiple design configuration descriptions.

10.2 designConfiguration

10.2.1 Schema

The following schema details the information contained in the **designConfiguration** element, which is one of the seven top-level elements of the schema.



10.2.2 Description

The **designConfiguration** element details the configuration for a design or generator chain description. The **designConfiguration** element contains the following mandatory and optional elements.

a) The ***versionedIdentifier*** group provides a unique identifier, made up of four subelements for a top-level IP-XACT element. See C.6.

b) **designRef** (mandatory) specifies the design description for this design configuration. The **designRef** element is of type ***libraryRefType*** (see C.7); it contains four attributes to specify the referenced VLNV.

c) **generatorChainConfiguration** (optional) is an unbounded list of configuration information associated with a **generatorChain** or a **generator** defined within a **generatorChain**. See 10.3.

d) **interconnectionConfiguration** (optional) is an unbounded list of information associated with interface interconnections. Any abstractors required for the connection of two interfaces are specified here. See 10.4.

e) **viewConfiguration** (optional) lists the active view for an instance of the design. It has the following subelements.

1) **instanceName** (mandatory) specifies the component instance name for which the view is being selected. This instance name shall be unique within the containing design configuration description. The **instanceName** element is of type ***Name***.

2) **viewName** (mandatory) defines the current valid view for the selected component instance. The **viewName** element is of type ***NMTOKEN***.

f) **description** (optional) allows a textual description of the design configuration. The **description** element is of type ***string***.

g) **vendorExtensions** (optional) adds any extra vendor-specific data related to the design configuration. See C.10.

See also: SCR 1.5, SCR 1.9, SCR 13.1, SCR 13.2, and SCR 13.4.

10.2.3 Example

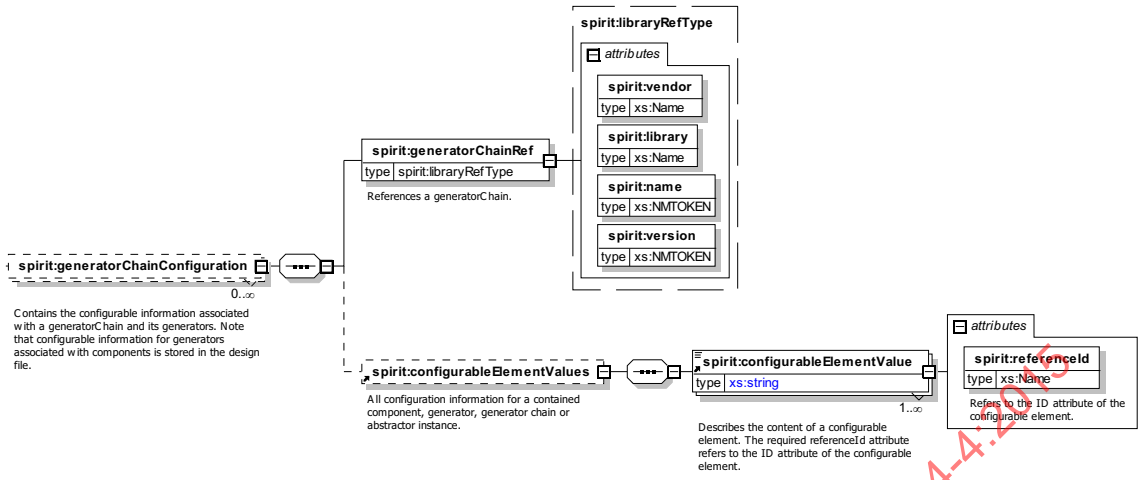
The following example shows a **designConfiguration** containing a generator chain configuration: one abstractor configuration in an **interconnectionConfiguration** and one instance view configuration.

```
<spirit:designConfiguration xmlns:spirit="http://www.spiritconsortium.org/
   XMLSchema/SPIRIT/1.5" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
   instance" xsi:schemaLocation="http://www.spiritconsortium.org/XMLSchema/
   SPIRIT/1.5/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:configurableElementValues>
            <spirit:configurableElementValue spirit:referenceId="tmpDir">
   my_temp_dir</spirit:configurableElementValue>
         </spirit:configurableElementValues>
   </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>
```

10.3 generatorChainConfiguration

10.3.1 Schema

The following schema defines information contained in **generatorChainConfiguration**, which may appear as an element inside the **designConfiguration** element.



10.3.2 Description

The **generatorChainConfiguration** element contains the configurable information associated with a **generatorChain** and its generators. It is up to the DE to decide how and when this configuration information is applied. Configurable information for any generators defined in a component or abstractor is stored in the design description with the associated instance's configuration. The **generatorChainConfiguration** element contains the following elements:

a) **generatorChainRef** (mandatory) specifies the generator chain description for this configuration information. The **generatorChainRef** element is of type ***libraryRefType*** (see C.7); it contains four attributes to specify the referenced **VLNV**.

b) **configurableElementValues** (optional) lists the generator chain's configurable parameter values. The **configurableElementValues** includes an unbounded list of **configurableElementsValue** elements.

   1) **configurableElementValue** (mandatory) is an unbounded list that specifies the value to apply to a configurable element; in this instance, it is pointed to by the **referenceId** attribute. The **configurableElementValue** is of type ***string***.

   2) The contained **referenceId** (mandatory) attribute is a reference to the **id** attribute of a configurable parameter value in the generator definition. The **referenceId** attribute is of type ***Name***.

See also: SCR 1.6, SCR 5.12, and SCR 13.6.

10.3.3 Example

The following example shows the configurable information for a **generatorChain**. Here parameters inside the referenced **generatorChain** are configured.

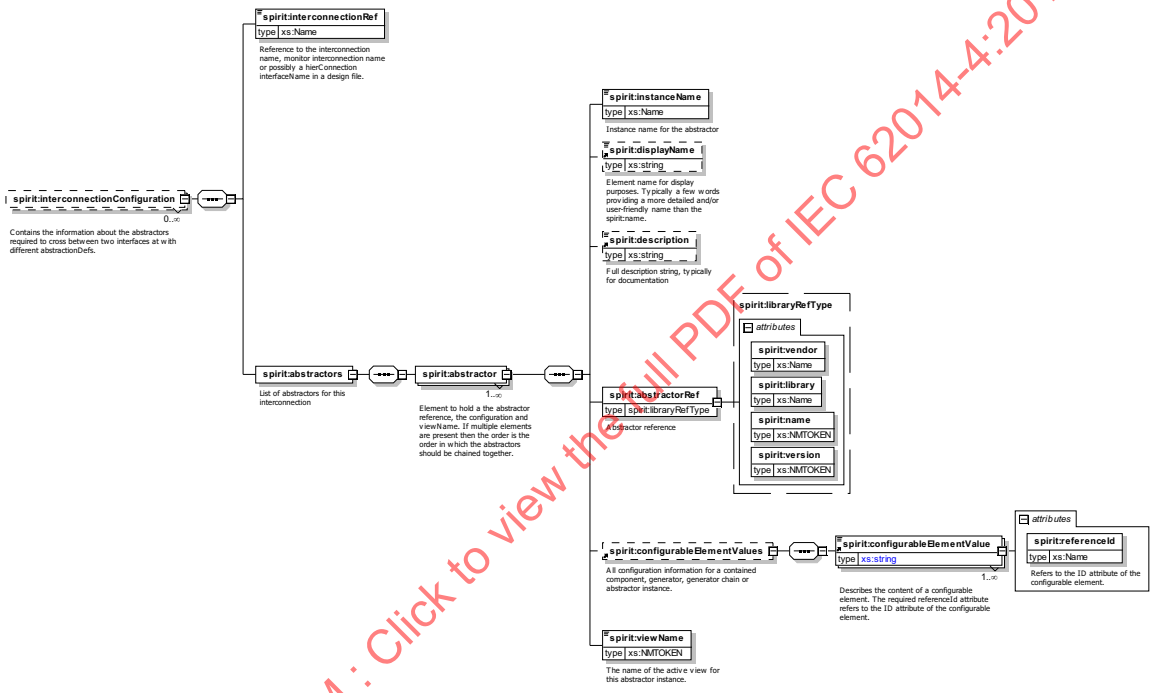
```
<spirit:generatorChainConfiguration>
   <spirit:generatorChainRef spirit:vendor="spiritconsortium.org"
   spirit:library="generatorLibrary" spirit:name="generator1"
   spirit:version="1.0"/>
      <spirit:configurableElementValues>
         <spirit:configurableElementValue spirit:referenceId="tmpDir">
   my_temp_dir</spirit:configurableElementValue>
         <spirit:configurableElementValue
   spirit:referenceId="verbose_level">1</spirit:configurableElementValue>
```

```
            <spirit:configurableElementValue spirit:referenceId="dump_log">
        true</spirit:configurableElementValue>
            </spirit:configurableElementValues>
    </spirit:generatorChainConfiguration>
```

10.4 interconnectionConfiguration

10.4.1 Schema

The following schema defines information contained in **interconnectionConfiguration** element, which may appear as an element inside the **designConfiguration** element.



10.4.2 Description

The **interconnectionConfiguration** element contains information about the **abstractor**s used to connect two interfaces having the same **busDefinition** types, but different **abstractionDefinition** types. The **interconnectionConfiguration** element contains the following elements.

a) **interconnectionRef** (mandatory) contains a reference to a design **interconnection**/**name** name, design **monitorInterconnection**/**name** name, or a design **hierConnection**/**interfaceRef** name. All **interconnectionRef** elements shall be unique within the containing design configuration description. The **interconnectionRef** element is of type ***Name***.

b) **abstractors** (mandatory) contains an unbounded list of **abstractor** elements. The list of **abstractor** elements is an ordered list for chaining the abstractors together to bridge from one abstraction to another. This element has the following subelements.

   1) **instanceName** (mandatory) assigns a unique name for this instance of the abstractor in this design. The value of this element shall be unique inside the **designConfiguration** and the referenced **design** element. The **instanceName** element is of type ***Name***.

   2) **displayName** (optional) allows a short descriptive text to be associated with the instance. The **displayName** is of type ***string***.

3) **description** (optional) allows a textual description of the instance. The **description** is of type ***string***.

4) **abstractorRef** (mandatory) is a reference to an abstractor description for this abstractor instance. The abstractorRef element is of type ***libraryRefType*** (see C.7); it contains four attributes to specify a unique VLNV.

5) **configurableElementValues** (optional) lists the abstractor instance's configurable parameter values. The **configurableElementValues** is an unbounded list of **configurableElementsValue** elements.

   i) **configurableElementValue** (mandatory) is an unbounded list that specifies the value to apply to a configurable element; in this instance, it is pointed to by the **referenceId** attribute. The **configurableElementValue** is of type ***string***.

   ii) The contained **referenceId** (mandatory) attribute is a reference to the **id** attribute of a configurable parameter value in the abstractor instance. The **referenceId** attribute is of type ***Name***.

6) **viewName** (mandatory) defines the current valid view for the selected abstractor instance. The **viewName** element is of type ***NMTOKEN***.

See also: SCR 1.12, SCR 3.7, SCR 3.8, SCR 3.9, SCR 3.10, SCR 3.11, SCR 3.12, SCR 3.13, SCR 3.14, SCR 3.15, SCR 5.13, SCR 13.3, and SCR 13.5.

10.4.3 Example

The following example shows the configuration of the `connection1` interconnection, with the definition of a chain of two abstractors to insert between the two component **busInterface**s. The abstractor instances are `abstraction1` and `abstraction2`, which convert from abstraction interface PV to PVT and PVT to RTL, respectively. The active views of these abstractor instances are `systemc` and `systemc_view`, respectively. 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>systemc</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>systemc_view</spirit:viewName>
      </spirit:abstractor>
   </spirit:abstractors>
</spirit:interconnectionConfiguration>
```

11. Addressing and data visibility

This clause describes how addresses are transformed between a slave's memory map and a master's address space. It also describes how to determine which bits of the memory map are visible in the master's address space.

The addressing descriptions here presume each bus interface only maps a single logical address port (a port with an **isAddress** qualifier) and a single logical data port (a port with an **isData** data qualifier). See also: 5.6 and 5.10.

If a bus interface maps more than one address or data port, then each combination of address and data ports implies a separate addressing and data visibility calculation. To calculate the address map for a particular type of transaction, the data and address ports that transaction uses need to be known first.

The most common case for multiple data ports in a single bus interface is where there are separate read and write data ports; however, their relevant properties of the read and write data ports are typically identical—giving identical read and write address maps.

11.1 Calculating the bit address of a bit in a memory map

A memory map consists of a set of address blocks, subspace maps, and banks containing further address blocks, subspace maps, and banks (to any number of levels). To calculate the address of a bit within an address block or subspace map relative to the containing memory map, its bit address needs to be calculated relative to its parent. If that parent is a bank, how that bank modifies the address needs to be calculated first, and then continue working up the bank structure until the memory map is reached. To do so, the following formulas apply.

— For a bit in an address block directly in a memory map:

$$memory\_map\_bit\_address = bit\_number\_in\_address\_block + addressBlock.baseAddress \times memoryMap.addressUnitBits \quad (1)$$

— For a bit in a subspace map:

$$memory\_map\_bit\_address = bit\_number\_in\_subspace\_map + subspaceMap.baseAddress \times memoryMap.addressUnitBits \quad (2)$$

For an address block or subspace map within a bank, the local bit address of a bit is simply its bit number. However, the following formulas need to be used on any containing banks.

a) For an item (bank, subspace map, or address block) within a serial bank:

$$item.rows = (item.range \times memoryMap.addressUnitBits) \div item.width \quad (3)$$

$$item.effective\_range = item.rows \times item.width \quad (4)$$

(i.e., the effective range of an item is its range rounded up to the nearest complete row)

The *item.range* of an item is calculated depending on its type:

1) For an address block or subspace map, the range is the value of the **range** subelement;
2) For a serial bank, the range is the sum of the effective ranges of the sub-items;
3) For a parallel bank, the range is the (largest *item.rows* of all the sub-items) × (*bank width*/*addressUnitBits*).

The *item.width* of an item is calculated depending on its type:

4) For an address block, the width is defined as the value of the **width** subelement;
5) For a subspace map, the width is the width of the address space of the referenced bus interface;
6) For a serial bank, the width is the width of the widest sub-item;
7) For a parallel bank, the width is the sum of the widths of the sub-items.

b) For a bit within item *n* in a serial bank:

$$parent\_bit\_address = child\_bit\_address + \sum_{i=1}^{n-1} item_i.effective\_range \times memoryMap.addressUnitBits \quad (5)$$

c) For a bit within item *n* in a parallel bank containing *m* items:

$$bit\_offset\_in\_row = child\_bit\_address \bmod item_n.width + \sum_{i=1}^{n-1} item_i.width \quad (6)$$

$$row\_offset = \sum_{i=1}^{m} item_i.width \times (child\_bit\_address \div (item_n.width)) \quad (7)$$

$$parent\_bit\_address = bit\_offset\_in\_row + row\_offset \quad (8)$$

Once the bit address within a top-level bank has been calculated, the bit address within the memory map can be derived from the following formula:

$$memory\_map\_bit\_address = bank\_bit\_address + bank.baseAddress \times memoryMap.addressUnitBits \quad (9)$$

11.2 Calculating the bus address at the slave bus interface

The bus address of a bit at the slave bus interface can be derived from the following formulas:

$$slave\_bus\_address = memory\_map\_bit\_address \div slave.bitsInLau \quad (10)$$

$$bus\_bit\_offset = memory\_map\_bit\_address \bmod slave.bitsInLau \quad (11)$$

On a bus, the bus address is the address carried by the address lines; the bit offset gives the offset within the LAU of the bit using the following formula:

$$slave\_bus\_address = (memory\_map\_bit\_address \div slave.bitsInLau) \,\&\, slave\_mapped\_address\_bits \quad (12)$$

where *slave_mapped_address_bits* is a mask derived from the set of address bits mapped in the slave.

11.3 Address modifications of an interconnection

The bus address is carried between adjacent bus interfaces (slave and mirrored slave, master and mirrored master, or master and slave) on the bus's **isAddress** logical port. If this port is a wire port, the address is

always carried as parallel bits with the least significant bit of the address on logical bit 0 of the port.[15] The interconnection can modify the address in two ways:

a) If some address bits are not connected, addresses with those bits set are not accessible from the master.
   1) Examine the logical vectors in the port maps to determine which address bits are connected.
   2) Transactional ports always carry all address bits across the interconnection.
b) If the value of **bitsInLau** differs on the two sides of the interconnection, the interpretation of the address as a bit address can vary by the ratio of the interfaces' **bitsInLau**. This, however, does not alter the actual bus address.

11.4 Address modifications of a channel

The address at the mirrored slave interface can be derived from the following formula:

$$mirrored\_slave\_bus\_address = slave\_bus\_address \;\&\; slave\_interconnection\_address\_bits \quad (13)$$

where *slave_interconnection_address_bits* is a mask derived from the set of address bits connected between the slave and the mirrored slave.

This is then modified by the remap address:

$$mirrored\_slave\_row\_address = mirrored\_slave\_bus\_address + \frac{mirroredSlave.baseAddress.remapAddress}{mirroredSlave.bitsInLau} \quad (14)$$

where *remapAddress* is the remap address for the current state of the channel.

How addresses are modified within a channel depends on the value of **bitSteering** in the mirrored slave interface. It also depends on the relative width of the mirrored master and mirrored slave data ports, where this width is defined to be the total number of bits of the logical data port that are mapped in the bus interface. If **bitSteering** is **true**, or the slave is wider than or the same width as the master, the addresses are simply modified to take into account any change in **bitsInLau** between the mirrored slave and the mirrored master, as shown in the following formula:

$$mirrored\_master\_bus\_address = floor\left(\frac{mirrored\_slave\_row\_address \times mirroredSlave.bitsInLau}{mirroredMaster.bitsInLau}\right) \;\&\; mirrored\_master\_mapped\_address\_bits \quad (15)$$

If **bitSteering** is **false** and the mirrored slave is narrower than the mirrored master, the address is adapted so all locations in the slave's memory map are visible:

$$mirrored\_slave\_bit\_address = mirrored\_slave\_row\_address \times mirroredSlave.bitsInLau \quad (16)$$

$$mirrored\_master\_bit\_address = mirrored\_slave\_bit\_address \bmod mirroredSlave.width + floor\left(\frac{mirrored\_slave\_bit\_address}{mirroredSlave.width}\right) \times mirroredMaster.width \quad (17)$$

[15] This gives a little-endian description of the address, which may differ from the address port description in the bus's documentation.

$$mirrored\_master\_bus\_address = floor\left(\frac{mirrored\_master\_bit\_address}{mirroredMaster.bitsInLau}\right) \quad (18)$$

& mirrored_master_mapped_address_bits

where *mirrored_master_mapped_address_bits* is a mask derived from the set of address bits mapped in the mirrored master port.

Finally, **bitSteering** has a different meaning in a mirrored slave interface than in a master or slave interface. In a master or slave interface, it means the component shall modify which bit lanes are used for data when accessing narrow devices. In a mirrored slave interface, it means the addresses from a mirrored master interface are not modified for transfers to a narrower mirrored slave data port.

11.5 Addressing in the master

The bus address at the master bus interface can be derived from the following formula:

$$master\_bus\_address = mirrored\_master\_bus\_address \ \&\ master\_interconnection\_address\_bits \quad (19)$$

where *master_interconnection_address_bits* is a mask derived from the set of address bits connected between the master and the mirrored master.

This gives a bit address of

$$master\_bit\_address = mirroredMaster.addressSpaceRef.baseAddress \times addressSpace.addressUnitBits + master\_bus\_address \times master.bitsInLau + bus\_bit\_offset \quad (20)$$

The bit address may then be converted to an addressing unit address and offset using the formulas:

$$address = master\_bit\_address \div addressSpace.addressUnitBits \quad (21)$$

$$offset = master\_bit\_address \bmod addressSpace.addressUnitBits \quad (22)$$

11.6 Visibility of bits

A bit in the slave's memory map is visible in the master's address space if:

— it is in an address range visible to the master;

— the master and slave agree on which bit lane the bit should appear and this bit lane is connected between the master and the slave.

11.6.1 Visible address ranges

Two conditions need to be fulfilled for an address in the slave to be visible to the master.

a) The address at the mirrored slave shall be within the range supported by the mirrored slave interface:

$$mirrored\_slave\_bus\_address < mirroredSlave.baseAddress.range \quad (23)$$

b) The address in the address space shall be within the range supported by the master address space for that bus interface:

$$0 \le master\_bit\_address < addressSpace.range \times addressSpace.addressUnitBits \quad (24)$$

11.6.2 Bit lanes in memory maps

The local bit lane of a bit in an address block is:

$$local\_bit\_lane = bit\_offset\_in\_address\_block \bmod addressBlock.width \quad (25)$$

Similarly, in a subspace map the bit lane is:

$$local\_bit\_lane = bit\_offset\_in\_subspace\_map \bmod subspaceMap.width \quad (26)$$

where the width of a subspace map is the width of the address space of the referenced master bus interface.

If the address block or subspace map is at the top-level of the memory map or only within serial banks, the bit lane in the memory map is the local bit lane.

If it is item *n* in a parallel bank, then:

$$bank\_bit\_lane = \sum_{i=1}^{n-1} item_i.width + local\_bit\_lane \quad (27)$$

If it is in multiple hierarchical parallel banks, this formula is applied at each higher level with the lower-level *bank_bit_lane* replacing *local_bit_lane*.

The bit lane in the memory map is the top-level *bank_bit_lane*.

11.6.3 Bit lanes in address spaces

The bit lane in an address space can be derived from the following formula:

$$address\_space\_bit\_lane = address\_space\_bit\_address \bmod addressSpace.width \quad (28)$$

11.6.4 Bit lanes in bus interfaces

In a bus interface, the logical bit numbers of the data port carry the corresponding bit lanes. For example, if a slave bus interface has a data port with a logical vector of `[15:8]`, this port can access bit lanes `15` to `8` of the memory map and logical bit lanes `15` to `8` in the connected mirrored slave or master interface.

11.6.5 Bit lanes in channels

All bus interfaces on a channel shall use the same logical numbering of data port bits. This means data bits cannot be moved between bit lanes in a channel by giving the mirrored bus interfaces different logical to physical mappings on their data ports.

11.6.6 Bit steering in masters and slaves

Bit steering only takes effect when the master and the slave have data ports of different widths. If they do and bit steering is enabled (i.e., **bitSteering** is **true** in the master or slave interface) for the bus interface with the wider data port, then this data port shall move its copy of output data to the correct bit lanes for the narrower port and read its input data from the correct bit lanes for the narrower port.

If bit steering is disabled in the wider port, the master can only access data at a particular address when the bit lane for that address in the address space is connected (through the bus interfaces and a channel) to the bit lane for the corresponding address in the memory map.

The following also apply.

— The **bitSteering** value has a different meaning in mirrored slaves. See 11.4.

— Some buses with bit steering may only support certain data port widths. Describing which widths are supported is outside the scope of IP-XACT.

— Bit steering allows software or hardware away from the bus interface to work without knowing the width of devices on the far side of the bus. To provide this functionality, a bus supporting bit steering normally gives the same address bits to all devices, irrespective of their widths, and does not adapt addresses to the width of the slave bus interfaces (i.e., **bitSteering** is **true** in the mirrored slave bus interfaces). Thus, a non-bit-steering master on such a bus only has access to some of the memory rows of narrower slaves.

11.7 Address translation in a bridge

The address at the master interface for a bridge can be derived from the following formulas:

a) The bus address at the master bus interface is:

$$master\_bus\_address = slave\_bus\_address \;\&\; master\_interconnection\_address\_bits \tag{29}$$

where *master_interconnection_address_bits* is a mask derived from the set of address bits connected between the master and the slave.

This gives a bit address of

$$master\_bit\_address = master\_bus\_address \times master.bitsInLau + master.addressSpaceRef.baseAddress \times addressSpace.addressUnitBits \tag{30}$$

The master bit address (also equal to the address space bit address) may be converted to an addressing unit address and offset of the **addressSpace** using the formulas:

$$address\_space\_address = master\_bit\_address \div addressSpace.addressUnitBits \tag{31}$$

$$address\_space\_offset = master\_bit\_address \bmod addressSpace.addressUnitBits \tag{32}$$

b) The bit address may also be converted to the address of the bridged slave interface by using the following formulas.

1) For a transparent bridge:

$$bridge\_slave\_address = (address\_space\_address \times addressSpace.addressUnitBits \div bridged\_slave.bitsInLau) \;\&\; bridged\_slave\_mapped\_address\_bits \tag{33}$$

2) For an opaque bridge:

$$bridge\_slave\_address = ((address\_space\_address - segmentAddressOffset) \times addressSpace.addressUnitBits \div (bridged\_slave.bitsInLau + bridge\_slave.baseAddress)) \;\&\; bridged\_slave\_mapped\_address\_bits \tag{34}$$

Annex A

(informative)

Bibliography

[B1] IEC/IEEE 61691-1-1, Behavioral languages—Part 1: VHDL language reference manual.[16, 17]

[B2] IEEE Std 754™-1985, IEEE Standard for Binary Floating-Point Arithmetic.[18]

[B3] IEEE Std 1364™, IEEE Standard for Verilog Hardware Description Language.

[B4] IEEE Std 1666™-2005, IEEE Standard for SystemC Language Reference Manual.

[B5] IETF RFC 2119, "Key words for use in RFCs to Indicate Requirement Levels," Bradner, S., Best Current Practice: 14.[19]

[B6] IP-XACT Leon Register Transfer Examples, v1.5.[20]

[B7] IP-XACT Leon Transaction Level Examples, v1.5.[21]

[B8] IP-XACT Schema, v1.5.[22]

[B9] IP-XACT Schema on-line documentation, v1.5.[23]

[B10] IP-XACT TGI WSDL, v1.5.[24]

[B11] ISO/IEC 8859-1, Information technology—8-bit single-byte coded graphic character sets—Part 1: Latin Alphabet No. 1.[25]

[B12] ISO/IEC 8879, Information processing—Text and office systems—Standard Generalized Markup Language (SGML).

[16]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/).

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

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

[19] Available from http://www.ietf.org/rfc/rfc2119.txt.

[20]Available from http://www.spiritconsortium.org/doc_downloads/.

[21]Available from http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5/index.xsd.

[22]See Footnote 21.

[23]See Footnote 21.

[24]Available from http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.5/TGI/TGI.wsdl.

[25]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/).

[B13] Transaction-Level Model of SystemC.[26]

[B14] IP-XACT standard tool names for **envIdentifier**.[27]

[26]Available at http://www.systemc.org.
[27]Available at http://www.spiritconsortium.org/tech/refs/toolnames.

Annex B

(normative)

Semantic consistency rules

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 clauses of this document, this annex defines them formally. Tools generating IP-XACT documents shall ensure these rules are obeyed. Tools reading IP-XACT documents shall report any breaches of these rules to the user.

B.1 Semantic consistency rule definitions

The following definitions apply when determining a semantic consistency rule (SCR) interpretation.

B.1.1 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.1.2 Interface mode of a bus interface

Specifies whether the bus interface is a **master**, **slave**, **system**, **mirroredMaster**, **mirroredSlave**, **mirroredSystem**, or **monitor** interface.

B.1.3 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.

B.1.4 Configurable element

Some elements in a component, abstractor, or generator chain description are defined as being configurable. See C.12.

NOTE—This is different from a **configurableElement** element, which is an element that references and sets the value of a configurable element.

B.1.5 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.

B.1.6 Memory mapping

If an access is not specified, the value defaults to the value from the level above. If the top level is not specified, the access defaults to read-write.

B.1.7 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 the following:

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.1.8 Logical and physical ports

a) If a wire **port** element in a component has a vector subelement, its range shall be [**left**:**right**], where **left** and **right** are the left and right values of the vector subelement. If it does not have a vector subelement, its range shall be [`0:0`].

b) If a **physicalPort** element has a vector subelement, its range shall be [**left**:**right**], where **left** and **right** are the left and right values of the vector subelement. If a **physicalPort** element does not have a vector subelement and it references a wire port, then its range shall be the range of the referenced model port.

c) If a **logicalPort** element has a vector subelement, its range shall be [**left**:**right**], where **left** and **right** are the left and right values of the vector subelement. If a **logicalPort** element does not have a vector subelement and the **physicalPort** element in the same **portMap** references a wire port, then its physical range shall be taken as [`abs(physical.left - physical.right):0`], where `physical.left` and `physical.right` are the left and right values of the physical port's range.

d) A logical bit of a port is mapped if it is included in the range of a **logicalPort** element referencing that bus interface port.

B.1.9 Addressable bus interface

A bus interface shall be *addressable* if the **isAddressable** element of the bus definition it references has the value **true**.

B.2 Rule listings

Most of the semantic rules listed here can be checked purely by manually examining a set of IP-XACT documents. A few, listed in Table B.14, need some external knowledge, so they cannot be checked this way. In Table B.1 through Table B.14, *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.

NOTE—Where these tables contain references to the values of elements and those elements are configurable in IP-XACT, then the values used are the configured values (not the XML element values).

B.2.1 Cross-references and VLNVs

Table B.1—Cross-references and VLNVs

Rule number	Rule	Single doc check	Notes
SCR 1.1	Every IP-XACT document visible to a tool shall have a unique VLNV.	No	Only applies 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. See also: C.6.2 and C.6.4.
SCR 1.2	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**. See also: C.6.2 and C.6.4.
SCR 1.3	The VLNV in an **extends** element in a bus definition shall be a reference to a bus definition.	No	See also: 5.2.2.
SCR 1.4	The VLNV in a **busType** element in a bus interface or abstraction definition shall be a reference to a bus definition.	No	See also: 6.5.1.
SCR 1.5	The VLNV in a **designRef** element in a design configuration shall be a reference to a **design**.	No	See also: 10.2.2.
SCR 1.6	The VLNV in a **generatorChainRef** element in a design configuration shall be a reference to a generator chain.	No	See also: 10.2.2 and 10.3.2.
SCR 1.7	The VLNV in a **generatorChainRef** subelement of the element **generatorChainSelector** in a generator chain shall be a reference to a generator chain.	No	See also: 9.2.2.
SCR 1.8	The VLNV in a **componentRef** element in a **design** shall be a reference to a **component**.	No	See also: 7.2.2.
SCR 1.9	The XML document element of an IP-XACT document shall be an **abstractor**, **abstractionDefinition**, **busDefinition**, **component**, **design**, **designConfiguration** or **generatorChain** element.	Yes	See also: 5.2.2, 5.3, 6.1.2, 7.1.2, 8.1.2, 9.1.2, and 10.2.2.
SCR 1.10	The VLNV in an **abstractionType** element in a component or abstractor shall reference an **abstractionDefiniton**.	No	See also: 8.1.2.
SCR 1.11	If a bus interface contains an **abstractionType** subelement, 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. See also: 5.2.2, 5.3.2, and 6.5.1.2.

Table B.1—Cross-references and VLNVs (*continued*)

Rule number	Rule	Single doc check	Notes
SCR 1.12	The VLNV in an **abstractorRef** in a **designConfiguration** shall reference an **abstractor**.	No	See also: 10.4.2.
SCR 1.13	The VLNV in an **extends** element in an abstraction definition shall be a reference to an abstraction definition.	No	See also: 5.3.2.

B.2.2 Interconnections

Table B.2—Interconnections

Rule number	Rule	Single doc check	Notes
SCR 2.1	In the attributes of an **activeInterface**, **monitoredActiveInterface** 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** and optional **path** attributes.	No	See also: 7.3.2 and 7.4.2.
SCR 2.2	In the subelements of an **interconnection**, the bus interfaces referenced by the two **activeInterface** subelements shall be compatible, i.e., the VLNVs of the **busType** elements within the two **busInterface** elements shall reference compatible **busDefinitions**.	No	See also: 6.3.1, 6.3.2, 6.3.3, and 7.3.2.
SCR 2.3	A particular component/bus interface combination shall appear in only one **interconnection** element in a design.	Yes	See also: 7.3.2.
SCR 2.4	An **interconnection** element shall only connect a master interface to a slave interface or a mirrored-master interface.	No	See also: 7.3.2.
SCR 2.5	An **interconnection** element shall only connect a mirrored-master interface to a master interface.	No	See also: 7.3.2.
SCR 2.6	An **interconnection** element shall only connect a slave interface to a master interface or a mirrored-slave interface.	No	See also: 7.3.2.
SCR 2.7	An **interconnection** element shall only connect a mirrored-slave interface to a slave interface.	No	See also: 7.3.2.
SCR 2.8	An **interconnection** element shall only connect a direct system interface to a mirrored-system interface.	No	See also: 7.3.2.
SCR 2.9	An **interconnection** element shall only connect a mirrored-system interface to a direct system interface.	No	See also: 7.3.2.

Table B.2—Interconnections (*continued*)

Rule number	Rule	Single doc check	Notes
SCR 2.10	In a direct master to slave connection, the value of **bitsInLAU** in the master's bus interface shall match the value of **bitsInLAU** in the slave's bus interface.	No	See also: 6.3.1 and 7.3.2.
SCR 2.11	In a direct master to slave connection, the **busDefinitions** referenced by the **busInterfaces** shall have a **directConnection** element with the value **true**.	No	See also: 6.3.1 and 7.3.2.
SCR 2.12	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	See also: 6.3.1, 6.3.2, 6.5.2.2, and 7.3.2.
SCR 2.13	If the same logical port is mapped in the port maps of both ends of a direct master to slave connection, then both bus interfaces shall map the same set of bits of that logical port.	No	Logical ports can only be identified with one another if the two bus interfaces reference the same abstraction definition. See also: 6.3.1 and 7.3.2.
SCR 2.14	The endianess in the two bus interfaces shall match for any interconnection using an addressable bus. If the endianess is not specified at either bus interface, it is presumed to be little endian.	No	See also: 6.3.1, 6.3.2, 6.5.1.2, and 7.3.2.
SCR 2.15	If a design contains a component with a **busInterface** that has a **connectionRequired** element with the value **true**, that **busInterface** shall be included in an **interconnection** of the design.	No	See also: 6.5.1.2 and 7.3.2.
SCR 2.16	A **monitorInterconnection** with interfaces that contain a **path** attribute with a **componetRef** and **busRef** shall exist in all hierarchical views.	No	See also: 7.4.

B.2.3 Channels, bridges, and abstractors

Table B.3—Channels, bridges, and abstractors

Rule number	Rule	Single doc check	Notes
SCR 3.1	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. See also: 5.3.2 and 6.6.2.
SCR 3.2	All bus interfaces referenced by a channel shall be mirrored interfaces.	Yes	See also: 6.4.1 and 6.6.2.

Table B.3—Channels, bridges, and abstractors (*continued*)

Rule number	Rule	Single doc check	Notes
SCR 3.3	A channel can be connected to no more mirrored-master **busInterface**s than the least value of **maxMasters** 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 **maxMasters**, as long as the bus definitions are compatible. See also: 6.6.2.
SCR 3.4	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. See also: 6.6.2.
SCR 3.5	Each bus interface on a component shall connect to only one channel of that channel component.	Yes	See also: 6.6.2.
SCR 3.6	The interface referenced by **masterRef** subelement of a **bridge** element shall be a master.	Yes	See also: 6.5.4.2.
SCR 3.7	The value of the **interconnectionRef** subelement of an **interconnectionConfiguration** element shall precisely match a design **interconnection**/**name**, a design **monitorInterconnection**/**name**, or a design **hierConnection**/**interfaceRef** of an interconnection described in the design referenced by the containing design configuration.	No	See also: 10.4.2.
SCR 3.8	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	See also: 10.4.2.
SCR 3.9	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	See also: 10.4.2.
SCR 3.10	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	See also: 10.4.2.
SCR 3.11	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	See also: 10.4.2.
SCR 3.12	An **interconnectionConfiguration** element shall not reference an interconnection in which the abstraction types referenced by the two endpoints are identical.	No	See also: 10.4.2.

Table B.3—Channels, bridges, and abstractors (*continued*)

Rule number	Rule	Single doc check	Notes
SCR 3.13	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	SCR 3.13 – SCR 3.15 mean the abstractors associated with an interconnection need to form a non-looping chain between the two ends. See also: 10.4.2.
SCR 3.14	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	See also: 10.4.2.
SCR 3.15	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	See also: 10.4.2.
SCR 3.16	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	See also: 5.3.2 and 8.1.2.
SCR 3.17	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 **extends** element referencing abstraction definition `AB` if bus definition `A` contains an **extends** element referencing bus definition `B`.	No	If abstraction definition `AA` extends abstraction definition `AB`, `AA` and `AB` need to be abstractions of different buses. See also: 5.3.2.
SCR 3.18	The interface referenced by the **masterRef** attribute of a **subspaceMap** element shall be a master interface that is bridged from a slave interface with a valid memory map referenced by its **memoryMapRef** element.	Yes	See also: 6.8.9.2.

B.2.4 Monitor interfaces and monitor interconnections

Table B.4—Monitor interfaces and monitor interconnections

Rule number	Rule	Single doc check	Notes
SCR 4.1	An **activeInterface** or **monitoredActiveInterface** element shall reference a **master**, **slave**, **system**, **mirroredMaster**, **mirroredSlave**, or **mirrored-System** interface.	No	See also: 6.3.3, 7.3.2, 7.4.2, and 7.6.2.
SCR 4.2	The **monitorInterface** subelements of a **monitorInterconnection** element shall reference a **monitor** bus interface.	No	See also: 6.3.3 and 7.3.2.
SCR 4.3	In a **monitorInterconnection** element, the value of the **interfaceMode** of the monitor interfaces shall match the mode of the **monitoredActiveInterface**.	No	This means all the monitor interfaces shall have the same interface mode. See also: 6.3.3, 6.5.2.2, and 7.3.2.
SCR 4.4	A **monitor** interface shall only be connected to a **system** or **mirroredSystem** interface if it has a **group** subelement and the value of this element matches the value of the **group** subelement of the **system** or **mirroredSystem** interface.	No	See also: 6.3.3, 6.5.2.2, and 7.3.2.
SCR 4.5	A particular **componentRef**/**busRef** 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. See also: 6.3.3 and 7.3.2.
SCR 4.6	All ports mapped in a **busInterface** with a mode of monitor shall have a direction of **in** for **wire** type ports or **requires** for **transactional** type ports.	Yes	See also: 6.3.3.

B.2.5 Configurable elements

Table B.5—Configurable elements

Rule number	Rule	Single doc check	Notes
SCR 5.1	A configurable element shall have a **dependency** attribute if and only if it has a **resolve** attribute with the value **dependent**.	Yes	See also: C.13 – C.17.
SCR 5.2	The value of a **dependency** attribute shall be an XPATH expression. This XPATH expression shall only reference items in the containing document.	Yes	See also: C.13 – C.17.
SCR 5.3	The XPATH expression in a **dependency** attribute shall not reference configurable elements having a **resolve** attribute value of **dependent** or **generated**.	Yes	See also: C.13 – C.17.
SCR 5.4	Any parameters used within all dependent parameter's `XPATH id()` calls shall exist.	Yes	See also: C.13 – C.17.
SCR 5.5	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. See also: C.13 – C.17.
SCR 5.6	An **id** attribute is required in any element with a **resolve** attribute value of **user** or **generated**.	Yes	See also: C.13 – C.17.
SCR 5.7	**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**. See also: C.13 – C.17.
SCR 5.8	**configurableElement** elements shall only reference configurable elements with a **resolve** attribute value of **user** or **generated**.	No	See also: C.13 – C.17.
SCR 5.9	If a **configurableElement** element references an element with a **formatType** attribute value of **float** or **long** and contains a **minimum** attribute, the value of the **configurableElementValue** element shall be greater or equal to the specified value of the **minimum** attribute.	No	See also: C.13 – C.17.
SCR 5.10	If a **configurableElement** element references an element with a **format** attribute value of **float** or **long** and contains a **maximum** attribute, the value of the **configurableElementValue** subelement shall be less than or equal to the specified value of the **maximum** attribute.	No	See also: C.13 – C.17.

Table B.5—Configurable elements (*continued*)

Rule number	Rule	Single doc check	Notes
SCR 5.11	If a **configurableElement** element references an element with a **choiceRef** attribute, the value for **configurableElementValue** subelement shall be one of the values listed in the **choice** element referenced by the **choiceRef** attribute.	No	See also: 6.14.2 and C.13 – C.17.
SCR 5.12	**configurableElement** elements within **generatorChainConfiguration** elements in design configuration documents elements shall only reference configurable elements that exist in the generator chain referenced by the **generatorChainRef** element; the value of the **referenceId** attribute of the **configurableElement** element shall match the value of the **id** attribute of some configurable element of a generator in the generator chain.	No	The schema guarantees uniqueness of **id** values in a generator chain. See also: 10.3.2.
SCR 5.13	**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. See also: 10.4.2.
SCR 5.14	A parameter's value or a configurable element's value shall match the element's **format** attribute.	See Note.	Yes for a parameter's value and No for a configurable element's value. See also: 7.2, C.13, C.14, C.15, C.16, and C.17.
SCR 5.15	A configurable element shall have a **bitStringLength** attribute if and only if it has a **format** attribute with the value **bitString**.	Yes	See also: C.13, C.14, C.15, C.16, and C.17.

B.2.6 Ports

Table B.6—Ports

Rule number	Rule	Single doc check	Notes
SCR 6.1	The value of the **name** subelement of any **logicalPort** element within a **busInterface** or **abstractorInterface** element shall match the value of a **logicalName** element of the abstraction definition referenced by the **busInterface** element.	No	This implies a bus interface that does not have an **abstractionType** element shall not contain a **portMaps** element, since there are no legal names for the **logicalPorts** within it. See also: 6.5.6.2.
SCR 6.2	If the abstraction definition referenced by a bus interface or abstractor interface specifies an initiative value for a logical transactional 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 or abstractor interfaces. For mirrored interfaces, the bus port initiative values needs to be reversed before doing the comparison.	No	See also: 5.11.2, 6.5.6.2, and 6.11.16.2.
SCR 6.3	If the abstraction bus definition referenced by a bus or abstractor interface specifies an initiative value for a logical transactional port of **provides** for that interface mode of bus or abstractor 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 or abstractor interfaces. For mirrored interfaces, the bus port initiative values shall be reversed before doing the comparison. Mirrored bus and abstractor interface shall be looked up as if they were not mirrored.	No	See also: 5.11.2, 6.5.6.2, and 6.11.16.2.

Table B.6—Ports (*continued*)

Rule number	Rule	Single doc check	Notes
SCR 6.4	If the abstraction definition referenced by a bus or abstraction interface specifies an initiative value for a logical transactional port of **both** for that interface mode of the bus or abstraction 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 **allLogical-InitiativesAllowed** 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 or abstraction interfaces. For mirrored interfaces, the bus port initiative values shall be reversed before doing the comparison. Mirrored bus and abstraction interfaces shall be looked up as if they were not mirrored.	No	See also: 5.11.2, 6.5.6.2, and 6.11.16.2.
SCR 6.5	If the abstraction definition referenced by a bus or abstraction interface specifies a direction for a logical wire 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	See also: 5.7.2, 6.5.6.2, 6.11.4.2, and and 8.6.2.
SCR 6.6	If the abstraction definition referenced by a bus or abstraction interface specifies a direction for a logical wire 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 or abstraction interfaces. For mirrored interfaces, the bus port directions shall be reversed before doing the comparison.	No	See also: 5.7.2, 6.5.6.2, 6.11.4.2, and and 8.6.2.

Table B.6—Ports (*continued*)

Rule number	Rule	Single doc check	Notes
SCR 6.7	If the abstraction definition referenced by a bus or abstraction interface specifies a direction for a logical wire 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 or abstraction interfaces. For mirrored interfaces, the bus port directions shall be reversed before doing the comparison.	No	See also: 5.7.2, 6.5.6.2, 6.11.4.2, and and 8.6.2.
SCR 6.8	If the abstraction definition referenced by a bus or abstraction 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 SCR 6.18. The third possible presence value (**optional**) neither forces nor forbids the inclusion of the port in the port map. See also: 5.11.2.
SCR 6.9	Only one component port in a port connection equivalence class may have the direction **out**.	No	See also: 7.3.2 and 7.5.4.
SCR 6.10	Only one component port in a port connection equivalence class may have the initiative **provides**.	No	See also: 7.3.2 and 7.5.5.
SCR 6.11	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	If a port in abstraction definition `B` is not used in bus interfaces using abstraction definition `A`, then, in abstraction definition `A`, that port shall have a presence value of **illegal** for all bus interface modes. See also: 5.3.2 and Table 2.
SCR 6.12	If the abstraction definition referenced by a bus or abstraction interface specifies a port is a wire port (i.e., the **port** element contains a **wire** subelement), the port map shall only map that logical port to a wire component port.	No	See also: 5.5.2, 6.5.6.2, 6.11.4.2, and and 8.6.2.
SCR 6.13	If the abstraction definition referenced by a bus or abstraction interface specifies a port is a transactional port (i.e., the **port** element contains a **transactional** subelement), the port map shall only map that logical port to a transactional component port.	No	See also: 5.10.2, 6.5.6.2, and 6.11.16.2.
SCR 6.14	At most one logical port of a port equivalence class shall be a port of a bus interface that participates in an interconnection to a bus interface using a different abstraction.	No	This rule prevents shared ports from crossing abstraction boundaries, since abstractors cannot describe the handling of such ports. See also: 5.10.2, 7.3.2, and 7.5.2.

Table B.6—Ports (*continued*)

Rule number	Rule	Single doc check	Notes
SCR 6.15	The value of the **group** subelement 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	See also: 5.5.2 and 5.10.2.
SCR 6.16	The value of the **group** subelement 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	See also: 6.5.2.2.
SCR 6.17	If an abstraction definition's **busType** element references an addressable bus, the abstraction definition shall contain at least one **port** element with an **isAddress** subelement.	No	See also: 5.2.2, 5.6.2, and 5.10.2.
SCR 6.18	If the abstraction definition referenced by a bus interface specifies, for a port, a presence value of **illegal** for that interface mode of bus or abstraction 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 or abstraction interfaces. Mirrored bus and abstraction interfaces shall be looked up as if they were not mirrored.	No	Port maps are optional, even on buses with required ports. See also SCR 6.8. The third possible presence value (**optional**) neither forces nor forbids the inclusion of the port in the port map. See also: 5.7.2 and 5.11.2.
SCR 6.19	The range of a **physicalPort** shall be a subset of the range of the referenced port in the component's model element.	Yes	See also: 6.5.6.2 and B.1.7.
SCR 6.20	Within any **portMap**, the sizes of the ranges of the **physicalPort** and the **logicalPort** shall be equal.	Yes	See also: 6.5.6.2.
SCR 6.21	If the abstraction definition port referenced by a **logicalPort** has a width defined, the upper limit of the range of the logical port shall be less than the width.	No	See also: 6.5.6.2.
SCR 6.22	Within a single bus interface no logical bit may be mapped more than once, i.e., if two or more **logicalPort** elements for that bus interface reference the same bus definition port, their ranges shall not overlap.	Yes	See also: 6.5.6.2.
SCR 6.23	If an abstraction definition port has a width defined, any bus interface containing a port map referencing that port needs to map all the bits of that port, i.e., every bit in the range `[width-1:0]` shall be mapped precisely once in the port maps of that bus interface.	No	This implies if there is only a single **logicalPort** referencing that bus port, its vector shall be `[width-1:0]` or `[0:width-1]`. See also: 6.5.6.2.

Table B.6—Ports (*continued*)

Rule number	Rule	Single doc check	Notes
SCR 6.24	If a transactional port in a component is mapped in a bus interface to a transactional port in an abstraction definition, then the set of names of **serviceTypeDef** elements in component port shall match the set of **typeNames** in the **ServiceType** element of the abstraction definition's port.	No	See also: 6.5.6.2 and 6.11.16.2.
SCR 6.25	Transactional ports shall only be connected together (by an ad hoc connection or through an interconnection) if neither of them contains a **serviceTypeDefs** element or they both contain identical **serviceTypeDefs** elements.	No	See also: 6.5.6.2 and 6.11.16.2.
SCR 6.26	A wire port with a direction of **out** shall not have a **driver** element.	Yes	See also: 6.11.6.2.
SCR 6.27	All ports referenced in an ad hoc connection shall have the same width, i.e., the absolute sizes of their ranges shall be identical.	No	See also: 7.5.4.

B.2.7 Registers

Table B.7—Registers

Rule number	Rule	Single doc check	Notes
SCR 7.1	No register shall have an **addressOffset** that falls within the address range of another register in the same address block, unless one of the registers and their **alternateRegisters** have non-conflicting **access** elements. Non-conflicting **access** elements have a value of **read-only**, **write-only**, or **writeOnce**. The address range of a register is the range `[addressOffset, addressOffset+ (size + addressBitUnits-1) ÷ addressBitUnits-1)*dim[n-1...0]`, where `dim` is the maximum number of elements for each of `n` dimensions.	Yes	I.e., registers shall not overlap, unless one is only visible when reading and the other is only visible when writing. See also: 6.10.2.2.
SCR 7.2	No bit field shall have a **bitOffset** value that falls within the bit range of another bit field, unless one of the registers has an **access** element with the value **read-only** and the other has an **access** element with the value **write-only** or **writeOnce**. The range of a bit field is the range `[bitOffset, bitOffset + width-1]`.	Yes	I.e., bit fields shall not overlap, unless one is only visible when reading and the other is only visible when writing. See also: 6.10.2.2 and 6.10.8.2.

Table B.7—Registers (*continued*)

Rule number	Rule	Single doc check	Notes
SCR 7.3	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 LAUs. This is equal to `(((size + addressBitUnits-1) ÷ addressBitUnits))* dim[n-1...0]`, where `dim` is the maximum number of elements for each of `n` dimensions.	Yes	See also: 6.10.2.2.
SCR 7.4	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	See also: 6.10.2.2 and 6.10.8.2.
SCR 7.5	The size of any register shall be no greater than the width of the containing address block.	Yes	See also: 6.8.6.2.
SCR 7.6	Any register in a register file shall fall entirely within that register file, i.e., for every `register 0 <= register.addressOffset < registerFileRange - registerSize`; where **registerFileRange** is the range of the register file and **registerSize** is the size of the register in LAUs. This is equal to `(((size + addressBitUnits-1) ÷ addressBitUnits))* dim[n-1...0]`, where `dim` is the maximum number of elements for each of `n` dimensions.	Yes	See also: 6.10.2, 6.10.3, and 6.10.6.
SCR 7.7	Any register file in an address block shall fall entirely within that address block, i.e., for every `register file 0 <= registerFile.addressOffset < addressBlockRange - registerFileSize`, where **registerBlockRange** is the range of the address block and **registerFileSize** is the size of the register in LAUs. This is equal to `registerFile.range * dim[n-1...0]`, where `dim` is the maximum number of elements for each of `n` dimensions.	Yes	See also: 6.8.2.
SCR 7.8	**volatile** cannot be set to **false** for an **addressBlock** where any containing **register** or **field** already has **volatile** set to **true**.	Yes	See also:6.10.2, 6.10.3, 6.10.8, 6.10.9, and 6.8.3.
SCR 7.9	**volatile** can not be set to **false** for a **register** where any containing **field** already has **volatile** set to **true**.	Yes	See also:6.10.2, 6.10.3, 6.10.8, and 6.10.9
SCR 7.10	When a **field** has **writeValueConstraint**/ **useEnumeratedValues** set to **true**, it also needs to have at least one **enumeratedValue** with the attribute **usage** set to **write** or **read-write**.	Yes	See also: 6.10.8, 6.10.9, and 6.10.10.

Table B.7—Registers (*continued*)

Rule number	Rule	Single doc check	Notes
SCR 7.11	When a **field** has a **writeValueConstraint**/**minimum** value and has a **writeValueConstraint**/**maximum** value, the value of **maximum** shall be greater than or equal to the value of **minimum**.	Yes	See also: 6.10.8, 6.10.9, and 6.10.10.
SCR 7.12	When multiple **field** elements have the same **typeIdentifier**, the field object shall contain the same contents for the elements in **fieldDefinitionGroup**.	Yes	See also: 6.10.8 and 6.10.9.
SCR 7.13	When multiple **register** or **alternateRegister** elements have the same **typeIdentifier**, the register object shall contain the same contents for the elements in the **registerDefinitionGroup** or **alternateRegisterDefinitionGroup**.	Yes	See also: 6.10.3 and 6.10.5.
SCR 7.14	When multiple **registerFile** elements have the same **typeIdentifier**, the register file object shall contain the same contents for the elements in the **registerFileDefinitionGroup**.	Yes	See also: 6.10.6.
SCR 7.15	When multiple **addressBlock** elements have the same **typeIdentifier**, the address block object shall contain the same contents for the elements in the **addressBlockDefinitionGroup**.	Yes	See also: 6.8.3.

B.2.8 Memory maps

Table B.8—Memory maps

Rule number	Rule	Single doc check	Notes
SCR 8.1	The width of an address block included in a memory map shall be a multiple of the memory map's **addressUnitBits**.	Yes	See also: 6.8.2.2.
SCR 8.2	Neither a parallel bank, nor banks within a parallel bank, shall contain subspace maps.	Yes	See also: 6.8.5.2, 6.8.7.2, and 6.8.8.2.
SCR 8.3	A **read-only bank** shall only contain **read-only addressBlock**s or **bank**s.	Yes	See also: 6.8.4.2 and 6.10.2.2.
SCR 8.4	A **read-only addressBlock** shall only contain **read-only register**s.	Yes	See also: 6.8.4.2 and 6.10.2.2.
SCR 8.5	A **read-only register** shall only contain **read-only field**s.	Yes	See also: 6.10.2.2.
SCR 8.6	A **write-only bank** shall only contain **write-only** or **writeOnce addressBlock**s or **bank**s.	Yes	See also: 6.8.4.2 and 6.10.2.2.
SCR 8.7	A **write-only addressBlock** shall only contain **write-only** or **writeOnce register**s.	Yes	See also: 6.8.4.2 and 6.10.2.2.

Table B.8—Memory maps (*continued*)

Rule number	Rule	Single doc check	Notes
SCR 8.8	A **write-only register** shall only contain **write-only** or **writeOnce field**s.	Yes	See also: 6.10.2.2.
SCR 8.9	A **register** shall only appear in an **addressBlock** of **usage register**.	Yes	See also: 6.8.4.2 and 6.10.2.2.
SCR 8.10	A **read-writeOnce bank** shall only contain **read-only**, **read-writeOnce**, or **writeOnce addressBlock**s or **bank**s.	Yes	See also: 6.8.4.2 and 6.10.2.2.
SCR 8.11	A **read-writeOnce addressBlock** shall only contain **read-only**, **read-writeOnce**, or **writeOnce register**s.	Yes	See also: 6.8.4.2 and 6.10.2.2.
SCR 8.12	A **read-writeOnce register** shall only contain **read-only**, **read-writeOnce**, or **writeOnce field**s.	Yes	See also: 6.10.2.2.
SCR 8.13	A **writeOnce bank** shall only contain **writeOnce addressBlock**s or **bank**s.	Yes	See also: 6.8.4.2 and 6.10.2.2.
SCR 8.14	A **writeOnce addressBlock** shall only contain **writeOnce register**s.	Yes	See also: 6.8.4.2 and 6.10.2.2.
SCR 8.15	A **writeOnce register** shall only contain **writeOnce field**s.	Yes	See also: 6.10.2.2.
SCR 8.16	Two **addressBlock** elements in the same **memoryMap** shall not overlap.	Yes	See also: 6.8.2.2.

B.2.9 Addressing

Table B.9—Addressing

Rule number	Rule	Single doc check	Notes
SCR 9.1	A non-hierarchical addressable master bus interface shall have an **addressSpaceRef** subelement.	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. See also: 5.6.2 and 6.5.3.
SCR 9.2	A non-hierarchical addressable slave bus interface shall have a **memoryMapRef** subelement or one or more **bridge** subelements 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. See also: 5.6.2 and 6.5.4.2.
SCR 9.3	Only an address space referenced by the **addressSpaceRef** subelement of a **cpu** element may contain an **exectutableImage** subelement.	No	See also: 6.7.1.2 and 6.7.3.2.

Table B.9—Addressing (*continued*)

Rule number	Rule	Single doc check	Notes
SCR 9.4	**bitSteering** is not allowed in mirrored-masters, system, or mirrored-system interface modes.	Yes	See also: 6.5.1.2.
SCR 9.5	Data widths in a channel shall all be a power 2 multiple of their **bitsInLau**.	Yes	See also: 6.5.1.2.
SCR 9.6	**bitsInLau** in a channel shall all be a power 2 multiple of the smallest **bitsInLau**.	Yes	See also: 6.5.1.2.
SCR 9.7	If a **languageTools** element contains a **linkerFlags** element or a **linkerCommandFile** element, it shall also contain a **linker** element.	Yes	See also: 6.7.4.2.
SCR 9.8	For each **segment** within an **addressSpace**, everything between **offsetAddress** and **offsetAddress** + **range** shall be contained within the range of that **addressSpace**.	Yes	See also: 6.7.1.2 and 6.7.2.2.
SCR 9.9	The **segmentRef** needs to reference an existing **segment** of the **addressSpace** in the master referenced by the **masterRef**.	Yes	See also: 6.8.9.2.

B.2.10 Hierarchy

Table B.10—Hierarchy

Rule number	Rule	Single doc check	Notes
SCR 10.1	All members of a hierarchical family of bus interfaces shall reference the same **busDefinition** in their **busType** subelements.	No	They need not reference the same abstraction definitions in their **abstractionType** elements. See also: 7.6.2.
SCR 10.2	All members of a hierarchical family of bus interfaces shall have the same interface mode (**master**, **slave**, **system**, etc.).	No	See also: 7.6.2.
SCR 10.3	If any member of a hierarchical family of bus interfaces has a **connectionRequired** element with a value of **true**, they all shall have this value.	No	See also: 7.6.2.
SCR 10.4	If any member of a hierarchical family of bus interfaces has a **bitSteering** element with a value of **true**, they all shall have this value.	No	See also: 7.6.2.
SCR 10.5	If any member of a hierarchical family of bus interfaces has a **portMap** subelement, they all shall.	No	See also: 7.6.2.
SCR 10.6	If any two bus interfaces in a hierarchical family of bus interfaces reference the same abstraction definitions, their **portMaps** shall also reference the same set of logical ports.	No	See also: 7.6.2.

Table B.10—Hierarchy (*continued*)

Rule number	Rule	Single doc check	Notes
SCR 10.7	In a hierarchical family of bus interfaces, all ports in the **portMap**s referencing the same bus port shall map the same set of bits from that logical port.	No	See also: 7.6.2.
SCR 10.8	In a hierarchical family of bus interfaces, the **physicalPort/name** of all ports in the **portMap** referencing the same logical port shall reference ports with the same direction.	No	See also: 7.6.2.
SCR 10.9	In a hierarchical family of bus interfaces, if the component ports referenced by the **physicalPort/name** of all ports in the **portMaps** referencing the same logical 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. See also: 7.6.2.
SCR 10.10	In a hierarchical family of bus interfaces, the **physicalPort/name** of all ports in the **portMap** referencing the logical bus port shall reference ports with identical **clockDriver** subelements.	No	See also: 7.6.2.
SCR 10.11	In a hierarchical family of bus interfaces, the **physicalPort/name** of all ports in the **portMap** referencing the same logical port shall reference ports with identical **singleShotDriver** subelements.	No	See also: 7.6.2.
SCR 10.12	In a hierarchical family of bus interfaces, the **physicalPort/name** of all ports in the **portMap** referencing the same logical port shall reference ports with identical **portConstraintSets** subelements.	No	See also: 7.6.2.

B.2.11 Hierarchy and memory maps

Table B.11—Hierarchy and memory maps

Rule number	Rule	Single doc check	Notes
SCR 11.1	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. See also: 7.6.2.
SCR 11.2	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. See also: 7.6.2.
SCR 11.3	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 SCR 11.4). See also: 7.6.2.
SCR 11.4	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. See also: 7.6.2.

B.2.12 Constraints

Table B.12—Constraints

Rule number	Rule	Single doc check	Notes
SCR 12.1	A component wire port with direction **out** shall not have a drive constraint.	Yes	See also: 6.11.11.2.
SCR 12.2	A component wire port with a direction **in** shall not have a load constraint.	Yes	See also: 6.11.12.2.

Table B.12—Constraints (*continued*)

Rule number	Rule	Single doc check	Notes
SCR 12.3	An **onMaster**, **onSlave**, or **onSystem** element of a wire port with direction **out** shall not contain a drive constraint within its **modeConstraint** element.	Yes	See also: 6.11.11.2.
SCR 12.4	An **onMaster**, **onSlave**, or **onSystem** element of a wire port with direction **in** shall not contain a load constraint within its **modeConstraint** element.	Yes	See also: 6.11.12.2.
SCR 12.5	An **onMaster**, **onSlave**, or **onSystem** element of a wire port with direction **out** shall not contain a load constraint within its **mirroredModeConstraint** element.	Yes	See also: 6.11.12.2.
SCR 12.6	An **onMaster**, **onSlave**, or **onSystem** element of a wire port with direction **in** shall not contain a drive constraint with its **mirroredModeConstraint** element.	Yes	See also: 6.11.11.2.
SCR 12.7	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	See also: 6.11.13.2.
SCR 12.8	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** subelement.	Yes	See also: 5.6.2 and 6.11.13.2.
SCR 12.9	The value of any **clockPeriod** element shall be greater than `0`.	Yes	See also: 6.11.7.2.
SCR 12.10	The value of any **clockPulseValue** element shall be `0` or `1`.	Yes	See also: 6.11.7.2.
SCR 12.11	The value of any **singleShotDuration** element shall be greater than `0`.	Yes	See also: 6.11.8.2.
SCR 12.12	The value of any **singleShotValue** element shall be `0` or `1`.	Yes	See also: 6.11.8.2.
SCR 12.13	Only a scalar port (i.e., a single-bit port) may have a **clockDriver** or a **singleShotDriver** subelement.	Yes	See also: 6.11.6.2.
SCR 12.14	A **whiteboxElementRef**, which references a **whiteboxElement** with a **whiteboxType** of **pin**, shall have a **pathName** that is a **port** in the containing description.	Yes	See also: 6.16.
SCR 12.15	A **whiteboxElementRef**, which references a **whiteboxElement** with a **whiteboxType** of **register**, shall have a **pathName** that is a **register** in the containing description.	Yes	See also: 6.16.

B.2.13 Design configurations

Table B.13—Design configurations

Rule number	Rule	Single doc check	Notes
SCR 13.1	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	See also: 10.2.2.
SCR 13.2	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** subelement of the **viewConfiguration** element.	No	See also: 10.2.2.
SCR 13.3	No two **interconnectionConfiguration** elements within a design configuration shall have the same **interconnectionRef** value.	Yes	See also: 10.4.2.
SCR 13.4	No two **viewConfiguration** elements within a design configuration shall reference the same view. i.e., no two **viewConfiguration** elements may have the same **instanceName**.	Yes	See also: 10.2.2.
SCR 13.5	No two **abstractor** elements within a design configuration shall have the same **instanceName** element values.	Yes	Also unique to the component instance names in the referenced design. See also: 10.4.2.
SCR 13.6	No two **generatorChainConfiguration** elements within the same design configuration shall reference the same generator chain through their **generatorChainRef** elements.	Yes	See also: 10.3.2.

B.2.14 Rules requiring external knowledge

Table B.14—Rules requiring external knowledge

Rule number	Rule	Single doc check	Notes
SCR 14.1	The **name** subelement of a **file** element can contain environment variables in the form of `${ENV_VAR}` that are meaningful to the host operating system and, when expanded, shall result in a string that is a valid URI.	Yes	See also: 6.13.2.2.
SCR 14.2	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`.	Yes	This is to guarantee uniqueness of vendor names. See also: C.6.2 and C.6.4.
SCR 14.3	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®`.[a] 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 Consortium Web site. See also: 6.11.2.2.

[a]Design Compiler and VCS are registered trademarks of Synopsys, Inc. This information is given for the convenience of users of this standard and does not constitute an endorsement by the IEEE of these products. Equivalent products may be used if they can be shown to lead to the same results.

Annex C

(normative)

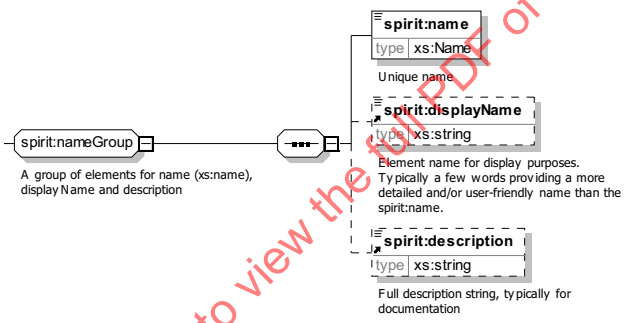
Common elements and concepts

This annex details common elements and concepts that appear many times throughout the standard.

C.1 nameGroup group

C.1.1 Schema

The following schema details the information contained in the ***nameGroup*** group.



C.1.2 Description

The ***nameGroup*** group defines any descriptive text for the containing element. The ***nameGroup*** group definition contains the following elements.

a) **name** (mandatory) identifies the containing element. The **name** element is of type ***Name***.

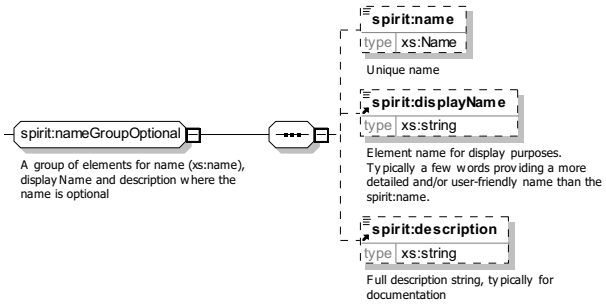
b) **displayName** (optional) allows a short descriptive text to be associated with the containing element. The **displayName** element is of type ***string***.

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

C.2 nameGroupOptional group

C.2.1 Schema

The following schema details the information contained in the ***nameGroupOptional*** group.



C.2.2 Description

The ***nameGroupOptional*** group defines any descriptive text for the containing element. The ***nameGroupOptional*** group definition contains the following elements.

a) **name** (optional) identifies the containing element. The **name** element is of type ***Name***.

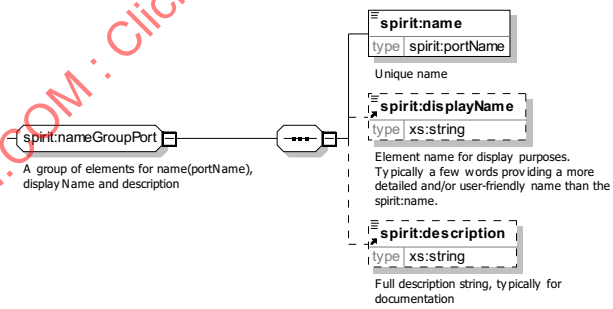
b) **displayName** (optional) allows a short descriptive text to be associated with the containing element. The **displayName** element is of type ***string***.

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

C.3 nameGroupPort group

C.3.1 Schema

The following schema details the information contained in the ***nameGroupPort*** group.



C.3.2 Description

The ***nameGroupPort*** group defines any descriptive text for the containing element. The ***nameGroupPort*** group definition contains the following elements.

a) **name** (mandatory) identifies the containing element. The **name** element is of type ***portName***.

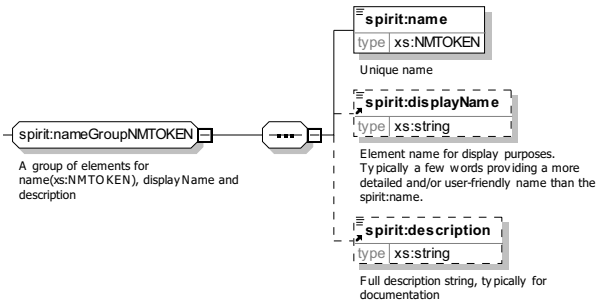
b) **displayName** (optional) allows a short descriptive text to be associated with the containing element. The **displayName** element is of type ***string***.

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

C.4 nameGroupNMTOKEN group

C.4.1 Schema

The following schema details the information contained in the ***nameGroupNMTOKEN*** group.



C.4.2 Description

The ***nameGroupNMTOKEN*** group defines any descriptive text for the containing element. The ***nameGroupNMTOKEN*** group definition contains the following elements.

a) **name** (mandatory) identifies the containing element. The name used shall match the corresponding port name found in any **views** of the containing component. The **name** element is of type ***NMTOKEN***.

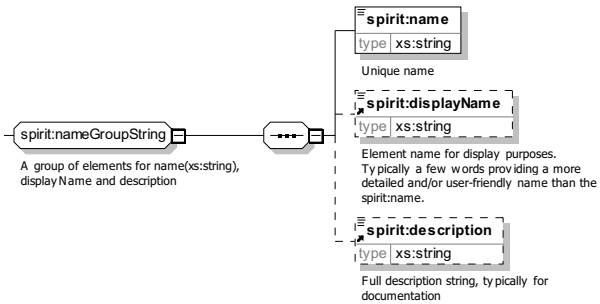
b) **displayName** (optional) allows a short descriptive text to be associated with the containing element. The **displayName** element is of type ***string***.

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

C.5 nameGroupString group

C.5.1 Schema

The following schema details the information contained in the ***nameGroupString*** group.



C.5.2 Description

The ***nameGroupString*** group defines any descriptive text for the containing element. The ***nameGroupString*** group definition contains the following elements.

a) **name** (mandatory) identifies the containing element. The **name** element is of type ***string***.

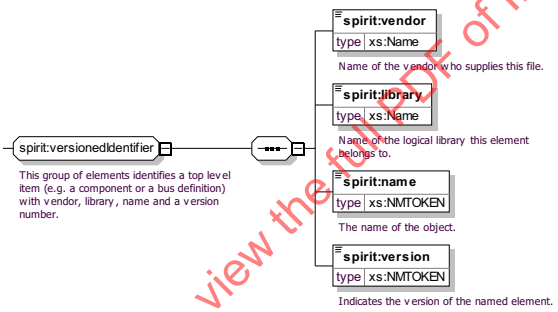
b) **displayName** (optional) allows a short descriptive text to be associated with the containing element. The **displayName** element is of type ***string***.

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

C.6 versionedIdentifier group

C.6.1 Schema

The following schema details the information contained in the ***versionedIdentifier*** group.



C.6.2 Description

The ***versionedIdentifier*** group defines a unique reference of or from an IP-XACT description. Only one object with a given VLNV may be present in a DE at any given time. The timing and way to change the VLNV of an object is completely up to the user or developer. The ***versionedIdentifier*** group definition contains the following four subelements.

a) **vendor** (mandatory) identifies the owner of this description. The format of the **vendor** element is the company internet domain name in left-to-right order (e.g., `spiritconsortium.org` not `org.spiritconsortium`). The **vendor** element is of type ***Name***.

b) **library** (mandatory) identifies the library of this description. This allows a vendor to group descriptions. The **library** element is of type ***Name***.

c) **name** (mandatory) identifies the name of this description within a library. The **name** element is of type ***NMTOKEN***.

d) **version** (mandatory) identifies the version of this description. This allows a vendor to provide many descriptions that all have the same name, but are still uniquely identified. The **version** 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. The **version** element is of type ***NMTOKEN***.

See also: SCR 1.1, SCR 1.2, and SCR 14.2.

C.6.3 Sorting and comparing version elements

Sorting and comparing **version** elements determines whether:

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

— multiple versions of the same IP exist in a design;

— a newer version of an IP exists.

To sort and compare **version** elements, 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.

C.6.3.1 Comparison rules

a) Each **version** element 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` and `A` are the same).

It is possible for different representations of version numbers to be considered equal. 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.

C.6.3.2 Comparison examples

The following examples illustrate the sorting and comparing of a **version** elements.

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

a) The first major field (`205`) is the same 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.

C.6.4 Version control

Each file conforming to the top-level schema has a set of VLNV elements that, 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 that might contain that information.

NOTE—An 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 may 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 a version.

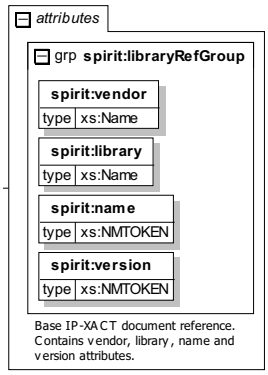
An IP-XACT description is one of the files that can be very usefully tracked in this way and updated in line with other design modifications. There is no direct link between the version number of the file and the VLNV identifier contained in that description. In many cases, however, the VLNV can be coordinated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

See also: SCR 1.1, SCR 1.2, and SCR 14.2.

C.7 libraryRefType

C.7.1 Schema

The following schema details the information contained in the ***libraryRefType*** type.



C.7.2 Description

The ***libraryRefType*** type defines a set of four attributes that reference another IP-XACT description through the unique VLNV identifier.

a) The **vendor** attribute (mandatory) identifies the owner of the referenced description. This attribute is of type ***Name***.

b) The **library** attribute (mandatory) identifies the library of the referenced description. This attribute is of type ***Name***.

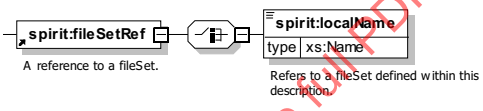
c) The **name** attribute (mandatory) identifies the name of the referenced description. This attribute is of type ***NMTOKEN***.

d) The **version** attribute (mandatory) identifies the version of the referenced description. This attribute is of type ***NMTOKEN***.

C.8 fileSetRef

C.8.1 Schema

The following schema details the information contained in the **fileSetRef** element.



C.8.2 Description

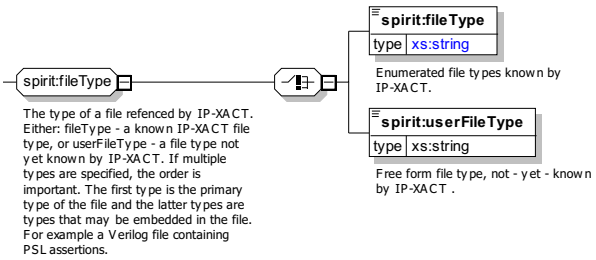
The **fileSetRef** element defines a reference to a **fileSet** contained in the containing document. The **fileSetRef** element contains the following element.

localName (mandatory) shall contain a name of a **fileSet**/**name** within the local description. **localName** is of type ***Name***.

C.9 fileType

C.9.1 Schema

The following schema details the information contained in the ***fileType*** type.



C.9.2 Description

The ***fileType*** type defines the format of a referenced file. The ***fileType*** group contains one or more of the following two elements.

a) **fileType** (mandatory) describes the type of file referenced from this enumerated list of industry standard files types.

1) **unknown**

2) **asmSource**, **cSource**, **cppSource**, **eSource**, **OVASource**, **perlSource**, **pslSource**, **SVASource**, **tclSource**, **veraSource**, **systemCSource**, **systemCSource-2.0**, **systemCSource-2.0.1**, **systemCSource-2.1**, **systemCSource-2.2**, **systemVerilogSource**, **systemVerilogSource-3.0**, **systemVerilogSource-3.1**, **systemVerilogSource-3.1a**, **verilogSource**, **verilogSource-95**, **verilogSource-2001**, **vhdlSource**, **vhdlSource-87**, and **vhdlSource-93**

3) **swObject** and **swObjectLibrary**

4) **vhdlBinaryLibrary** and **verilogBinaryLibrary**

5) **executableHdl** and **unelaboratedHdl**

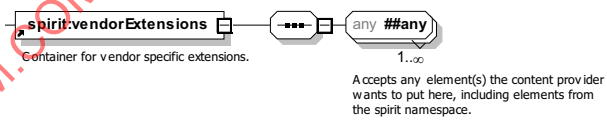
6) **SDC**

b) **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.10 vendorExtensions

C.10.1 Schema

The following schema details the information contained in the **vendorExtensions** element.



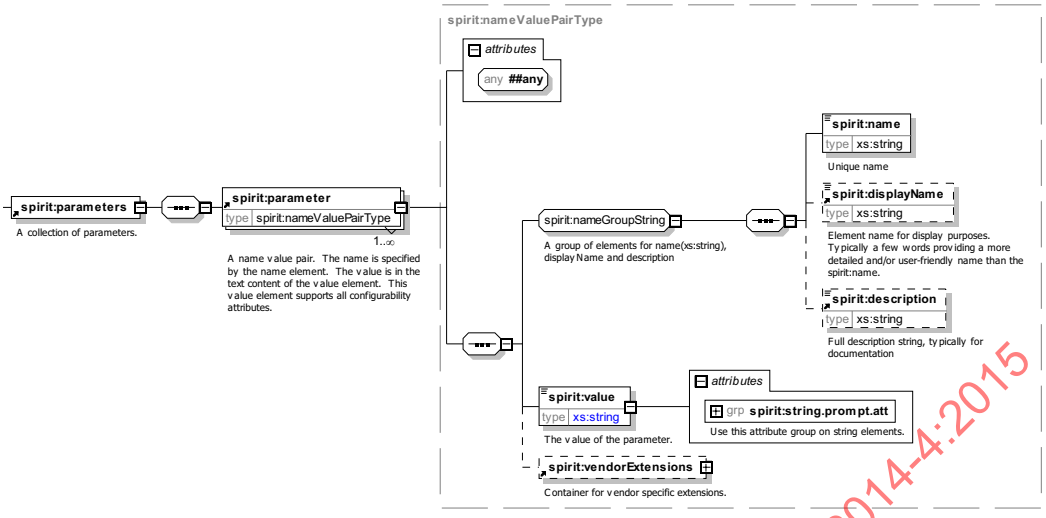
C.10.2 Description

The **vendorExtensions** element is a place in the description in which any vendor specific information can be stored. The **vendorExtensions** element allows any well-formed description.

C.11 parameters

C.11.1 Schema

The following schema details the information contained in the **parameters** element.



C.11.2 Description

The **parameters** element contains an unbounded list of **parameter** elements. **parameter** (mandatory) defines a configurable element related to the containing element. The parameter definition allows for the assignment of a name and a value. The **parameter** element also allows for vendor attributes to be applied. The **parameter** element definition contains the following elements.

a) ***nameGroupString*** is defined in C.1.

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 C.12.

c) **vendorExtensions** (optional) adds any extra vendor-specific data related to the **parameter**. See C.10.

C.12 Configuration

Some elements in a component, abstractor, or generator chain description are defined as being configurable. This means that the value of the element can be set differently for each use of the description, which allows a single description to be used in many different ways.

The same method is used to configure elements in a component, abstractor, or generator chain. The configuration is done via a reference to an **id** attribute in the configured element. Any element in a component, abstractor, or generator chain description with an **id** attribute is configurable. The **id** attribute is always contained inside an attribute group. This group provides other attributes that specify how the element may be configured, e.g., from a choice list or free-form text. This group also defines the type of values for setting the element, e.g., **integer**, **float**, or **string**. There are five different attribute groups defined for four different types of configurable elements:

a) bool.prompt.att, see C.13.

b) float.prompt.att, see C.14.

c) long.prompt.att or long.att, see C.15 or C.16, respectively.

d) string.prompt.att, see C.17.

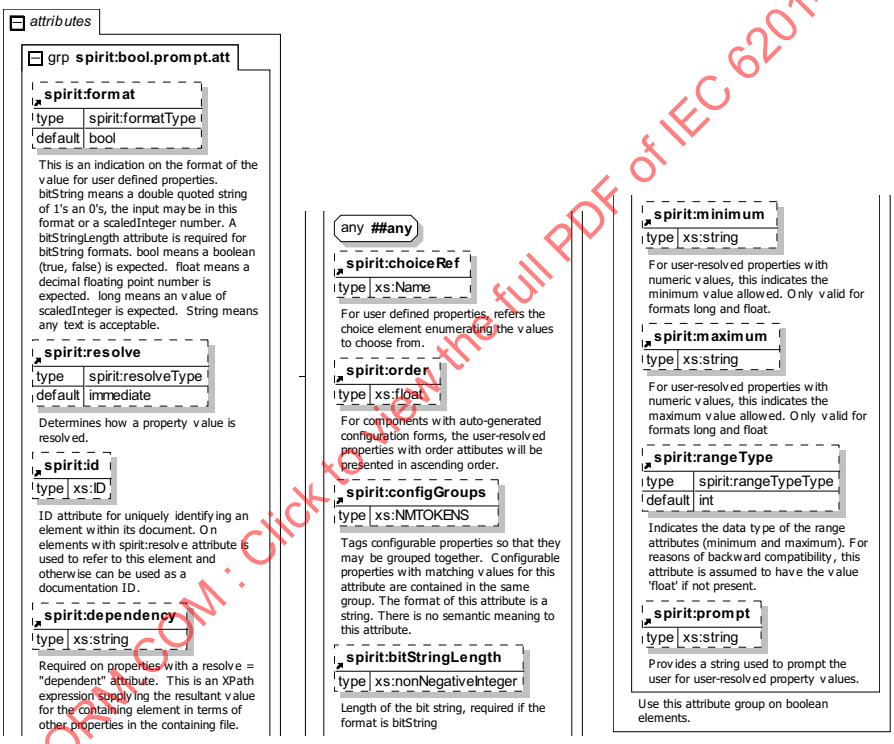
The location of the configuration values differs based on the description being configured. A component description is configured via the design description, see Clause 7. When the component instance is

referenced in the design description, a **configurableElementValue** element may be specified to configure any elements for this instance, see 7.2. An abstractor or generator chain description is configured via the design configuration description, see 10.2. A design configuration description may contain an **interconnectionConfiguration** element or a **generatorChainConfiguration** element, each of which may contain a **configurableElementValue** element used to configure an abstractor or generator chain, respectively.

C.13 bool.prompt.att

C.13.1 Schema

The following schema details the information contained in the ***bool.prompt.att*** attribute group.



C.13.2 Description

The ***bool.prompt.att*** attribute group defines a set of attributes to be applied to the containing element. The ***bool.prompt.att*** attribute group contains the following attributes.

a) **format** (optional) is the input and storage type for the configurable elements and, optionally, the output type for **modelParameters**. The value shall be one from of the following.

   1) **bitString** indicates the input or storage shall be in the format of a double quoted string of `1`'s and `0`'s or a ***scaledInteger*** number. The output shall be a bit string in the format specified by the language, e.g., `VHDL = "1010"` or `Verilog = 4'b1010`.

   2) **bool** (the default) indicates the input or storage shall be one of **true** or **false**. The output shall be a ***boolean*** type as specified by the language.

3) **float** indicates the input or storage shall a decimal floating point number. The output shall be a decimal floating point number.

4) **long** indicates the input shall be a ***scaledInteger*** number. The storage shall be in a format compatible with the containing element and the output shall be a decimal integer number.

5) **string** indicates the input and storage shall contain any characters. The output shall be a string in the format as specified by the language.

b) **resolve** (optional) defines how the value for the containing element is configured. The value shall be one of the following.

1) **immediate** (the default) indicates the value shall be specified in the containing element.

2) **user** indicates the value shall be specified by user input and the new value stored in a design or design configuration description under the **configurableElement** element.

3) **dependent** indicates the value shall be defined by an XPATH equation (see Annex E) defined in the **dependency** attribute. The **dependency** attribute requires the **resolve** attribute to be equal to **dependent**.

4) **generated** indicates the value shall be set by a generator and the new value stored in a design or design configuration description under the **configurableElement** element.

c) **id** (optional) assigns a unique identifier to the containing element for reference throughout the containing description. The **id** attribute is required if the element has a **resolve** type equal to **user**, **generated,** or is referenced in a **dependency** equation. This **id** is referenced in two ways. The first reference is by the **configurableElement** in a design or design configuration description. The second is in a **dependency** attribute XPATH equation. The **id** attribute if of type ***ID***.

d) **dependency** (optional) is an XPATH 1.0 equation (see Annex E) for the value of the containing element. The **resolve** attribute shall be equal to **dependent**. The **dependency** attribute is of type ***string***.

e) **##any** (optional) indicates any additional attributes in any namespace are allowed in the containing element. These additional attributes are called *vendor attributes*.

f) **choiceRef** (optional) indicates the value of the containing element is defined in the referenced **choice** element. The **choiceRef** attribute is of type ***Name***.

g) **order** (optional) indicates how elements are presented, when **resolve** equals **user**. The elements are presented in ascending order. The **order** attribute is of type ***float***.

h) **configGroups** (optional) indicates a name to group elements together; elements with matching values for this attribute are contained in the same group. There is no semantic meaning to this attribute. The **configGroups** attribute is of type ***NMTOKENS***.

i) **bitStringLength** (optional) indicates the length of the bit string. The **format** attribute shall be equal to **bitString**. The **bitStringLength** attribute is of type ***nonNegativeInteger***.

j) **minimum** (optional) indicates the lower bound for the value of the containing element. This check is only valid for a format of **bitString**, **float**, or **long**. The **rangeType** attribute shall specify the type of the **minimum** attribute. The **minimum** attribute is of type ***string***.

k) **maximum** (optional) indicates the upper bound for the value of the containing element. This check is only valid for a format of **bitString**, **float**, or **long**. The **rangeType** attribute shall specify the type of the **maximum** attribute. The **maximum** attribute is of type ***string***.

l) **rangeType** (optional) indicates the range for the **minimum** and **maximum** attributes. The value shall be one from the enumerated list of **float** (the default), **int**, **unsigned int**, **long**, or **unsigned long**. **float** indicates a floating point number, **int** or **long** indicates a ***scaledInteger***, and **unsigned int** or **unsigned long** indicates a ***scaledNonNegativeInteger***.

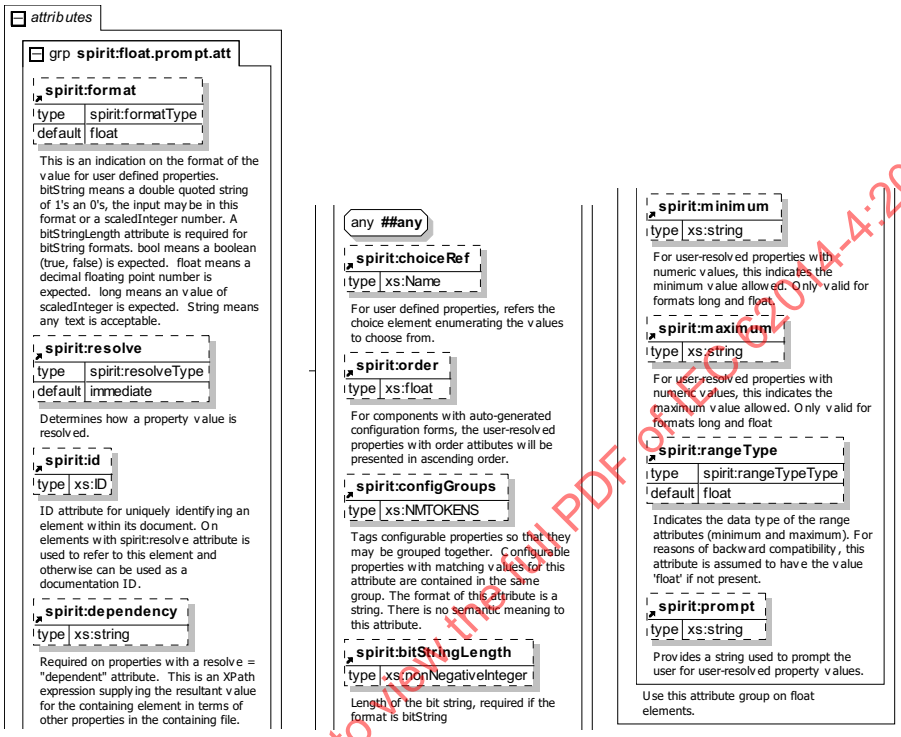
m) **prompt** (optional) defines a prompt string that a DE can use if the **resolve** attribute is equal to **user**. The **prompt** attribute is of type ***string***.

See also: SCRs in Table B.5.

C.14 float.prompt.att

C.14.1 Schema

The following schema details the information contained in the ***float.prompt.att*** attribute group.



C.14.2 Description

The ***float.prompt.att*** attribute group defines a set of attributes to be applied to the containing element. The ***float.prompt.att*** attribute group contains the following attributes.

a) **format** (optional) is the input and storage type for the configurable elements and, optionally, the output type for **modelParameters**. The value shall be one from of the following.

   1) **bitString** indicates the input or storage shall be in the format of a double quoted string of `1`'s and `0`'s or a ***scaledInteger*** number. The output shall be a bit string in the format specified by the language, e.g., `VHDL = "1010"` or `Verilog = 4'b1010`.

   2) **bool** indicates the input or storage shall be one of **true** or **false**. The output shall be a ***boolean*** type as specified by the language.

   3) **float** (the default) indicates the input or storage shall a decimal floating point number. The output shall be a decimal floating point number.

   4) **long** indicates the input shall be a ***scaledInteger*** number. The storage shall be in a format compatible with the containing element, and the output shall be a decimal integer number.

   5) **string** indicates the input and storage shall contain any characters. The output shall be a string in the format as specified by the language.

b) **resolve** (optional) defines how the value for the containing element is configured. The value shall be one from the enumerated list of **immediate**, **user**, **dependent**, or **generated**.

1) **immediate** (the default) indicates the value shall be specified in the containing element.
2) **user** indicates the value shall be specified by user input and the new value stored in a design or design configuration description under the **configurableElement** element.
3) **dependent** indicates the value shall be defined by an XPATH equation (see Annex E) defined in the **dependency** attribute. The **dependency** attribute requires the **resolve** attribute to be equal to **dependent**.
4) **generated** indicates the value shall be set by a generator and the new value stored in a design or design configuration description under the **configurableElement** element.

c) **id** (optional) assigns a unique identifier to the containing element for reference throughout the containing description. The **id** attribute is required if the element has a **resolve** type equal to **user**, **generated,** or is referenced in a **dependency** equation. This **id** is referenced in two ways. The first reference is by the **configurableElement** in a design or design configuration description. The second is in a **dependency** attribute XPATH equation. The **id** attribute if of type ***ID***.

d) **dependency** (optional) is an XPATH 1.0 equation (see Annex E) for the value of the containing element. The **resolve** attribute shall be equal to **dependent**. The **dependency** attribute is of type ***string***.

e) **##any** (optional) indicates any additional attributes in any namespace are allowed in the containing element. These additional attributes are called *vendor attributes*.

f) **choiceRef** (optional) indicates the value of the containing element is defined in the referenced **choice** element. The **choiceRef** attribute is of type ***Name***.

g) **order** (optional) indicates how elements are presented, when **resolve** equals **user**. The elements are presented in ascending order. The **order** attribute is of type ***float***.

h) **configGroups** (optional) indicates a name to group elements together; elements with matching values for this attribute are contained in the same group. There is no semantic meaning to this attribute. The **configGroups** attribute is of type ***NMTOKENS***.

i) **bitStringLength** (optional) indicates the length of the bit string. This attribute is required if the **format** attribute is equal to **bitString**. The **bitStringLength** attribute is of type ***nonNegativeInteger***.

j) **minimum** (optional) indicates the lower bound for the value of the containing element. This check is only valid for a format of **bitString**, **float**, or **long**. The **rangeType** attribute shall specify the type of the **minimum** attribute. The **minimum** attribute is of type ***string***.

k) **maximum** (optional) indicates the upper bound for the value of the containing element. This check is only valid for a format of **bitString**, **float**, or **long**. The **rangeType** attribute shall specify the type of the **maximum** attribute. The **maximum** attribute is of type ***string***.

l) **rangeType** (optional) indicates the range for the **minimum** and **maximum** attributes. The value shall be one from the enumerated list of **float** (the default), **int**, **unsigned int**, **long**, or **unsigned long**. **float** indicates a floating point number, **int** or **long** indicates a ***scaledInteger***, and **unsigned int** or **unsigned long** indicates a ***scaledNonNegativeInteger***.

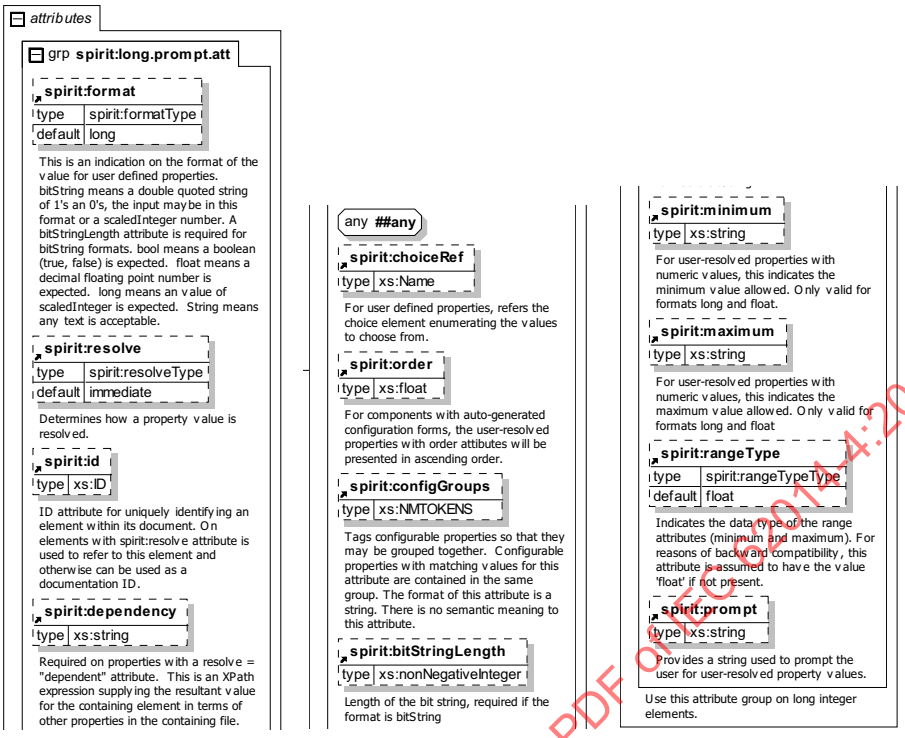
m) **prompt** (optional) defines a prompt string that a DE can use if the **resolve** attribute is equal to **user**. The **prompt** attribute is of type ***string***.

See also: SCRs in Table B.5.

C.15 long.prompt.att

C.15.1 Schema

The following schema details the information contained in the ***long.prompt.att*** attribute group.



C.15.2 Description

The ***long.prompt.att*** attribute group defines a set of attributes to be applied to the containing element. The ***long.prompt.att*** attribute group contains the following attributes.

a) **format** (optional) is the input and storage type for the configurable elements and, optionally, the output type for **modelParameters**. The value shall be one from of the following.

   1) **bitString** indicates the input or storage shall be in the format of a double quoted string of `1`'s and `0`'s or a ***scaledInteger*** number. The output shall be a bit string in the format specified by the language, e.g., `VHDL = "1010"` or `Verilog = 4'b1010`.

   2) **bool** indicates the input or storage shall be one of **true** or **false**. The output shall be a ***boolean*** type as specified by the language.

   3) **float** indicates the input or storage shall a decimal floating point number. The output shall be a decimal floating point number.

   4) **long** (the default) indicates the input shall be a ***scaledInteger*** number. The storage shall be in a format compatible with the containing element and the output shall be a decimal integer number.

   5) **string** indicates the input and storage shall contain any characters. The output shall be a string in the format as specified by the language.

b) **resolve** (optional) defines how the value for the containing element is configured. The value shall be one from the enumerated list of **immediate**, **user**, **dependent,** or **generated**.

   1) **immediate** (the default) indicates the value shall be specified in the containing element.

   2) **user** indicates the value shall be specified by user input and the new value stored in a design or design configuration description under the **configurableElement** element.

3) **dependent** indicates the value shall be defined by an XPATH equation (see Annex E) defined in the **dependency** attribute. The **dependency** attribute requires the **resolve** attribute to be equal to **dependent**.

4) **generated** indicates the value shall be set by a generator and the new value stored in a design or design configuration description under the **configurableElement** element.

c) **id** (optional) assigns a unique identifier to the containing element for reference throughout the containing description. The **id** attribute is required if the element has a **resolve** type equal to **user**, **generated** or is referenced in a **dependency** equation. This **id** is referenced in two ways. The first reference is by the **configurableElement** in a design or design configuration description. The second is in a **dependency** attribute XPATH equation. The **id** attribute if of type ***ID***.

d) **dependency** (optional) is an XPATH 1.0 equation (see Annex E) for the value of the containing element. The **resolve** attribute shall be equal to **dependent**. The **dependency** attribute is of type ***string***.

e) **##any** (optional) indicates any additional attributes in any namespace are allowed in the containing element. These additional attributes are called *vendor attributes*.

f) **choiceRef** (optional) indicates the value of the containing element is defined in the referenced **choice** element. The **choiceRef** attribute is of type ***Name***.

g) **order** (optional) indicates how elements are presented, when **resolve** equals **user**. The elements are presented in ascending order. The **order** attribute is of type ***float***.

h) **configGroups** (optional) indicates a name to group elements together; elements with matching values for this attribute are contained in the same group. There is no semantic meaning to this attribute. The **configGroups** attribute is of type ***NMTOKENS***.

i) **bitStringLength** (optional) indicates the length of the bit string. This attribute is required if the **format** attribute is equal to **bitString**. The **bitStringLength** attribute is of type ***nonNegativeInteger***.

j) **minimum** (optional) indicates the lower bound for the value of the containing element. This check is only valid for a format of **bitString**, **float**, or **long**. The **rangeType** attribute shall specify the type of the **minimum** attribute. The **minimum** attribute is of type ***string***.

k) **maximum** (optional) indicates the upper bound for the value of the containing element. This check is only valid for a format of **bitString**, **float**, or **long**. The **rangeType** attribute shall specify the type of the **maximum** attribute. The **maximum** attribute is of type ***string***.

l) **rangeType** (optional) indicates the range for the **minimum** and **maximum** attributes. The value shall be one from the enumerated list of **float** (the default), **int**, **unsigned int**, **long**, or **unsigned long**. **float** indicates a floating point number, **int** or **long** indicates a ***scaledInteger***, and **unsigned int** or **unsigned long** indicates a ***scaledNonNegativeInteger***.

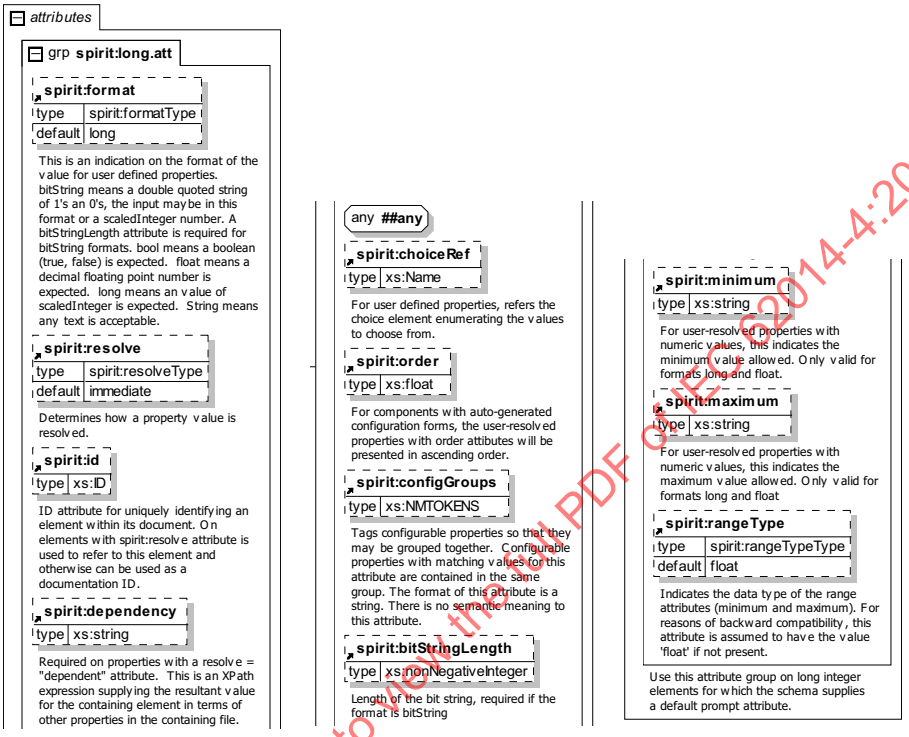
m) **prompt** (optional) defines a prompt string that a DE can use if the **resolve** attribute is equal to **user**. The **prompt** attribute is of type ***string***.

See also: SCRs in Table B.5.

C.16 long.att

C.16.1 Schema

The following schema details the information contained in the ***long.att*** attribute group.



C.16.2 Description

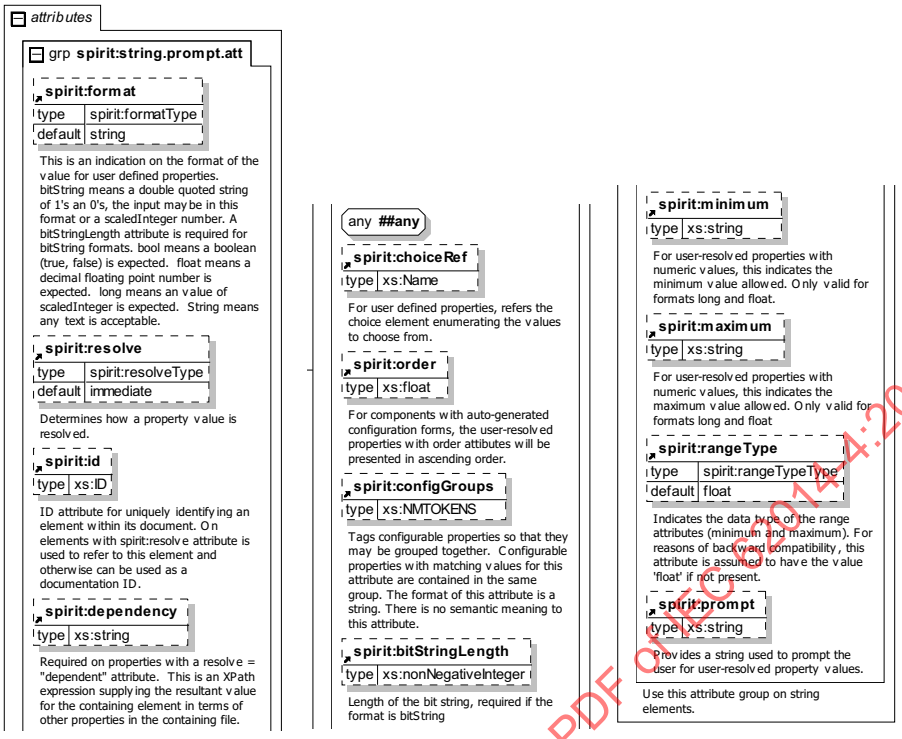
The ***long.att*** attribute group defines a set of attributes to be applied to the containing element. The ***long.att*** attribute group contains all the same attributes as the ***long.prompt.att*** attribute group, except for the **prompt** attribute. See C.15.

See also: SCRs in Table B.5.

C.17 string.prompt.att

C.17.1 Schema

The following schema details the information contained in the ***string.prompt.att*** attribute group.



C.17.2 Description

The ***string.prompt.att*** attribute group defines a set of attributes to be applied to the containing element. The ***string.prompt.att*** attribute group contains the following attributes.

a) **format** (optional) is the input and storage type for the configurable elements and, optionally, the output type for **modelParameters**. The value shall be one from of the following.

   1) **bitString** indicates the input or storage shall be in the format of a double quoted string of `1`'s and `0`'s or a ***scaledInteger*** number. The output shall be a bit string in the format specified by the language, e.g., `VHDL = "1010"` or `Verilog = 4'b1010`.

   2) **bool** indicates the input or storage shall be one of **true** or **false**. The output shall be a ***boolean*** type as specified by the language.

   3) **float** indicates the input or storage shall a decimal floating point number. The output shall be a decimal floating point number.

   4) **long** indicates the input shall be a ***scaledInteger*** number. The storage shall be in a format compatible with the containing element and the output shall be a decimal integer number.

   5) **string** (the default) indicates the input and storage shall contain any characters. The output shall be a string in the format as specified by the language.

b) **resolve** (optional) defines how the value for the containing element is configured. The value shall be one from the enumerated list of **immediate**, **user**, **dependent,** or **generated**.

   1) **immediate** (the default) indicates the value shall be specified in the containing element.

   2) **user** indicates the value shall be specified by user input and the new value stored in a design or design configuration description under the **configurableElement** element.

   3) **dependent** indicates the value shall be defined by an XPATH equation (see Annex E) defined in the **dependency** attribute. The **dependency** attribute requires the **resolve** attribute to be equal to **dependent**.

4) **generated** indicates the value shall be set by a generator and the new value stored in a design or design configuration description under the **configurableElement** element.

c) **id** (optional) assigns a unique identifier to the containing element for reference throughout the containing description. The **id** attribute is required if the element has a **resolve** type equal to **user**, **generated,** or is referenced in a **dependency** equation. This **id** is referenced in two ways. The first reference is by the **configurableElement** in a design or design configuration description. The second is in a **dependency** attribute XPATH equation. The **id** attribute if of type ***ID***.

d) **dependency** (optional) is an XPATH 1.0 equation (see Annex E) for the value of the containing element. The **resolve** attribute shall be equal to **dependent**. The **dependency** attribute is of type ***string***.

e) **##any** (optional) indicates any additional attributes in any namespace are allowed in the containing element. These additional attributes are called *vendor attributes*.

f) **choiceRef** (optional) indicates the value of the containing element is defined in the referenced **choice** element. The **choiceRef** attribute is of type ***Name***.

g) **order** (optional) indicates how elements are presented, when **resolve** equals **user**. The elements are presented in ascending order. The **order** attribute is of type ***float***.

h) **configGroups** (optional) indicates a name to group elements together; elements with matching values for this attribute are contained in the same group. There is no semantic meaning to this attribute. The **configGroups** attribute is of type ***NMTOKENS***.

i) **bitStringLength** (optional) indicates the length of the bit string. This is required if the **format** attribute is equal to **bitString**. The **bitStringLength** attribute is of type ***nonNegativeInteger***.

j) **minimum** (optional) indicates the lower bound for the value of the containing element. This check is only valid for a format of **bitString**, **float**, or **long**. The **rangeType** attribute shall specify the type of the **minimum** attribute. The **minimum** attribute is of type ***string***.

k) **maximum** (optional) indicates the upper bound for the value of the containing element. This check is only valid for a format of **bitString**, **float**, or **long**. The **rangeType** attribute shall specify the type of the **maximum** attribute. The **maximum** attribute is of type ***string***.

l) **rangeType** (optional) indicates the range for the **minimum** and **maximum** attributes. The value shall be one from the enumerated list of **float** (the default), **int**, **unsigned int**, **long**, or **unsigned long**. **float** indicates a floating point number, **int** or **long** indicates a ***scaledInteger***, and **unsigned int** or **unsigned long** indicates a ***scaledNonNegativeInteger***.

m) **prompt** (optional) defines a prompt string that a DE can use if the **resolve** attribute is equal to **user**. The **prompt** attribute is of type ***string***.

See also: SCRs in Table B.5.

Annex D

(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.

D.1 boolean

The **boolean** type defines two possible values, **true** and **false**.

D.2 configurableDouble

The **configurableDouble** type defines a decimal floating point number based on the IEEE single-precision 64-bit floating point type (see IEEE Std 754-1985 [B2]).

D.3 float

The **float** type defines a decimal floating point number based on the IEEE single-precision 32-bit floating point type (see IEEE Std 754-1985 [B2]).

D.4 ID or IDREF

The **ID** or **IDREF** type defines a unique identifier through the containing description. It needs to begin with a letter or underscore (_). An **ID** or **IDREF** shall only contain letters, numbers, and the underscore (_), dash (-), and dot (.) characters. Any leading or trailing spaces are removed.

D.5 instancePath

The **instancePath** type defines a series of **Name** type character strings, see D.8, separated by a slash (/). Any leading or trailing space is removed.

D.6 integer

The **integer** type defines a decimal integer number of infinite precision, containing the numbers 0–9.

D.7 libraryRefType

The **libraryRefType** type is an element type, not a data type. This type defines the four attributes of a VLNV required for a reference from one description to another description. See C.7.

D.8 Name

The **Name** type defines a series of any characters, excluding embedded whitespace. It needs to begin with a letter, colon (`:`), or underscore (_). A **Name** shall only contain letters, numbers, and the colon (`:`), underscore (_), dash (−), and dot (`.`) characters. Any leading or trailing spaces are removed.

D.9 NMTOKEN

The **NMTOKEN** type defines a series of any characters, excluding embedded whitespace. It shall only contain letters, numbers, and the colon (`:`), underscore (_), dash (−), and dot (`.`) characters. Any leading or trailing spaces are removed.

D.10 NMTOKENS

The **NMTOKENS** type defines a series of any characters, including embedded whitespace. It shall only contain letters, numbers, and the colon (`:`), underscore (_), dash (−), and dot (`.`) characters.

D.11 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`.

D.12 portName

The **portName** type defines a series of any characters, excluding embedded whitespace. It shall only contain letters, numbers, and the colon (`:`), underscore (_), dash (−), and dot (`.`) characters. It also needs to begin with a letter, colon (`:`), or underscore (_). Any leading or trailing spaces are removed.

D.13 positiveInteger

The **positiveInteger** type is a subtype of **integer**; it follows all the same rules, except its value shall be greater than `0`.

D.14 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`.

D.15 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`.

D.16 scaledPositiveInteger

The **scaledPositiveInteger** type is a subtype of **scaledInteger**; it follows all the same rules, except its value shall be greater than `0`.

D.17 SpiritURI

The **SpiritURI** type defines a string of characters for an absolute or relative path to a file, a directory, or an executable in URI format (`xs:anyURI`), except it can contain environment variables in the `${ENV_VAR}` form, which are replaced by their value(s) to provide the underlying URI.

D.18 string

The **string** type defines a series of any characters and may include spaces.

D.19 token

The **token** type defines a series of any characters, excluding carriage-return, line-feed, and tab. Any leading or trailing spaces are removed and all internal sequences of two or more spaces are reduced to one space.

Annex E

(normative)

Dependency XPATH

This 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, a **dependency** attribute is required.

The accuracy of the XPATH numeric functions shall be of infinite precision and are not limited to any fixed number of bits. This is necessary to ensure all systems are interoperable and the large calculations required by the configuration of IP-XACT components are successful.

In addition to the standard XPATH 1.0 functions, IP-XACT also defines the following functions to aid expressions calculations.

E.1 id

```
id(string)
```

The **id** function returns the value of the element with an attribute of **id** that matches the input *string*. This function has been modified from the standard XPATH definition to return the value applied to the element at the time of evaluation; this is the configured value of the element from the design description (see G.4).

E.2 spirit:containsToken

```
spirit:containsToken(string1, string2)
spirit:containsToken(node, stringx)
```

The **containsToken** function (boolean) returns *true* if *string1* contains *string2* as a token (or *node* contains *stringx* as a token) and otherwise returns *false*. To be interpreted as a token, *string2* needs to be found within *string1* (or *stringx* needs to be found within *node*) and be separated by whitespace from any other characters in the *string1* (or *node*) that are not whitespace characters.

containsToken only uses the configured value while executing its function.

Purpose: Some attributes in IP-XACT are a list of tokens separated by whitespace. 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.

E.3 spirit:decode

```
spirit:decode(string)
spirit:decode(node)
```

The **decode** function (number) decodes the string (or node) argument to a number and returns the number or NaN [if the string (or node) cannot be decoded]. If the string (or node) 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. `M` is a multiplier of 1024×1024. `G` is a multiplier of 1024×1024×1024. `T` is 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.

E.4 spirit:pow

```
spirit:pow(number, number)
spirit:pow(number, node)
spirit:pow(node, number)
spirit:pow(node, node)
```

The **pow** function (number) returns a number (or node), 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.

E.5 spirit:log

```
spirit:log(number, number)
spirit:log(number, node)
spirit:log(node, number)
spirit:log(node, node)
```

The **log** function (number) returns a number (or node), which 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.

E.6 Dependency example

This is a example of using **resolve=dependent**.

```
<spirit:memoryMaps>
  <spirit:memoryMap>
    <spirit:name>mmap</spirit:name>
    <spirit:addressBlock>
       <spirit:name>ab1</spirit:name>
      <spirit:baseAddress  spirit:resolve="user" spirit:id="baseAddr">0</
   spirit:baseAddress>
<spirit:range spirit:id="range">786K</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('baseAddr'))+ spirit:decode(id('range')))+1))"
    spirit:id="dependentBaseAddress">0</spirit:baseAddress>
<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 F

(informative)

External bus with 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 that interconnects the internal and external bus. Some interface 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.

F.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 F.1.

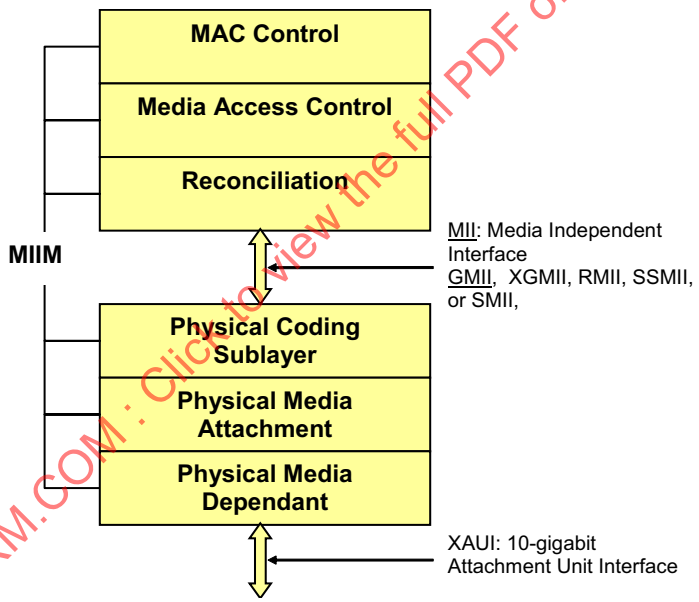


Figure F.1—Ethernet interface examples

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 pin-out. The reduction in ports is

achieved by multiplexing data and control information to a port transmit port and a single receive port.

F.2 Example: I²C bus

The I²C 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 internal digital interface shown in Figure F.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`.

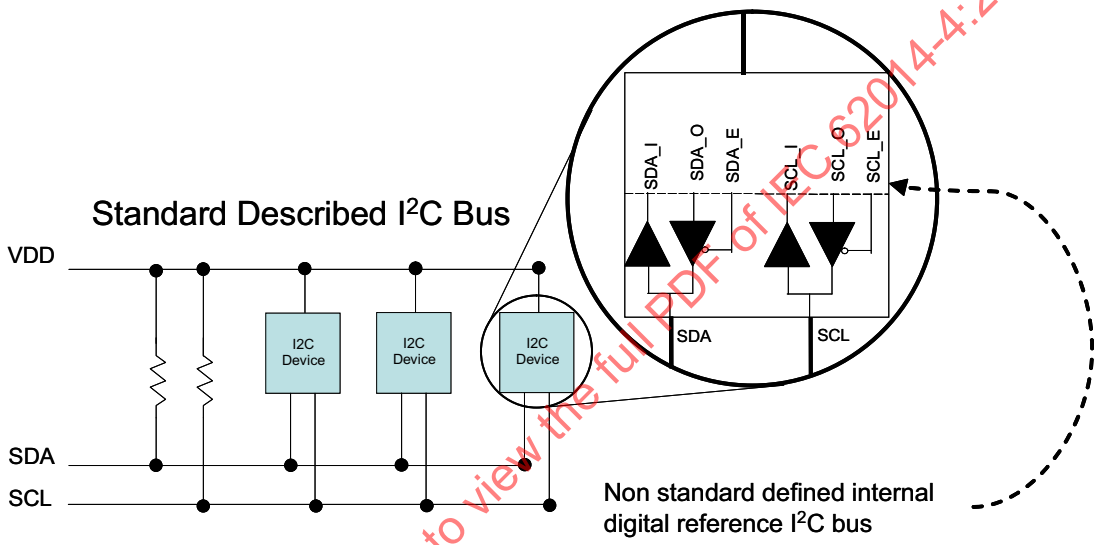


Figure F.2—I²C interface example

Annex G

(normative)

Tight generator interface

IP-XACT generators are tools that are invoked from within a DE to perform an operation required by the user of the DE. For example, generators can be provided to verify the configuration of a subsystem, generate an address map, or write a netlist representation of the subsystem in a target language such as Verilog or SystemC. To perform their various operations, most generators need access to the IP-XACT meta-data describing the subsystem, as currently loaded into the DE. Generators need both read- and write-access to the IP-XACT meta-data. All generators are external applications running in a separate address space from the DE.

The TGI defines how the DE and generator cooperate to achieve the desired end-goal of the user of the DE. The TGI defines the method of communication between the DE and generator, the method for invoking the generator, and the actual application programming interface (API) that can be used to read and write the IP-XACT meta-data stored in the DE. G.1, G.2, and G.3 describe each of these three aspects of the TGI, respectively.

G.1 Method of communication

The DE and the generator communicate with each other by sending messages to each other utilizing the SOAP standard. SOAP provides a simple means for sending XML-format messages using HTTP or other transport protocols. The TGI restricts the set of allowed transport protocols to HTTP and a file-based protocol. All generators are required to support the HTTP protocol, but support for the file-based protocol is optional. The same rules apply to the DE—it shall support the use of the HTTP protocol, but is not required to support the file-based protocol, even though a generator may allow it. The protocols supported by a generator are specified using the **transportMethod** element within the **componentGenerator** element.

The information required to use a particular transport protocol shall be passed to the generator by the DE when it is invoked, as described in G.2. For the HTTP protocol, the generator is passed a URL of the form **http://***host_name***:***port_number*. All SOAP messages sent to the DE shall be sent using the referenced URL. For the file-based protocol, the generator is passed a URL of the form **file://***file_name*. In this case, all SOAP messages are written to the specified file.

Each DE and generator is responsible for setting itself up to communicate using SOAP with the appropriate transport protocol. For example, a generator written in Tcl might include the Tcl SOAP package to enable SOAP functionality. Once the communication channel is set up, the generator can read and write the IP-XACT meta-data using any legal SOAP message. The set of legal SOAP messages defines the API portion of the TGI (see G.7).

G.2 Generator invocation

All of the information known by the DE about a particular generator comes from an instance of the **componentGenerator** (see 6.12), **abstractorGenerator** (see 8.7), or **generator** (see 9.4) elements. These elements provides the following information.

a) **name** is the name of the generator as seen within the DE.

b) **executable** is the URL defining the location of the generator.

c) **parameters** is a list of name/value pairs defining information to be passed to the generator.
d) **apiType** indicates the generator type: **TGI** or **none** (no communication).
e) **transportMethods** show any transport mechanisms supported (in addition to HTTP).
f) **phase** (not relevant to the TGI).
g) **vendorExtensions** (not relevant to the TGI).
h) **group** (not relevant to the TGI).

G.2.1 Resolving the URL

The URL defining the generator executable shall resolve to one of the following forms.

— **file:***path_to_executable* (e.g., `file:/usr/jdoe/bin/mygen.pl` or `file:./bin/mygen.pl`) defines the path for invoking the generator on the machine from which the DE was invoked.
— **file://***machine_name*/*path_to_executable* (e.g., `file://server1/tmp/othergen.pl`) defines the path for invoking a generator on the specified machine.
— **http://***web_address*:*port_number* (e.g., `http://www.acme.com/generator:1500`) defines the URL of a generator implemented as a Web-based server.

All *file references* are relative to the location of the XML description in which the file reference is contained.

For the file-based generators, the DE shall invoke the generator as a sub-process with a command line built up as:

executable **-url** *transport_URL generator_parameter_arguments*

The *generator_parameter_arguments* are the parameters from the **componentGenerator** element with the user-specified values. Each parameter causes two additional arguments to be passed to the generator with the following format: **-***parameter_name parameter_value*. The *transport_URL* is created by the DE, but is guaranteed to specify a protocol supported by the generator as defined by the transport methods within the **componentGenerator**. The DE is responsible for ensuring any passed parameters can be interpreted correctly. This URL is to be used in the generator to set up the SOAP communication channel.

For Web-based generators, the DE shall send a message to the address and port defined as the executable. The format of this message is

url=transport_URL&*generator_parameter_arguments*

In this case, the generator parameters are formatted using the standard HTTP parameter passing syntax. The specified transport URL shall be used by the generator for any return messages to the DE.

The invocation syntax described above applies only to generators with an API type of **TGI**. Generators with an API type of **none** are invoked as described above, excluding the **transport_URL** argument.

G.2.2 Example

This example shows file-based and Web-based **componentGenerator** elements.

```
<spirit:componentGenerator>
      <spirit:name>myGenerator</spirit:name>
      <spirit:parameter spirit:name="param1" spirit:resolve="user"
   spirit:id="param1">default1</spirit:parameter>
      <spirit:parameter spirit:name="param2">fixedValue</spirit:parameter>
```

```
        <spirit:apiType>TGI</spirit:apiType>
        <spirit:transportMethods>
            <spirit:transportMethod>file</spirit:transportMethod>
        </spirit:transportMethods>
        <spirit:generatorExe>../bin/myGenerator.pl</spirit:generatorExe>
    </spirit:componentGenerator>
```

produces the following output.

```
path_to _XML/../bin/myGenerator -url http://host:port -param1 default1
    -param2 fixedValue
```

Whereas:

```
    <spirit:componentGenerator>
        <spirit:name>myWebGenerator</spirit:name>
        <spirit:parameter spirit:name="param" spirit:resolve="user"
    spirit:id="myParamID">defaultValue</spirit:parameter>
        <spirit:apiType>TGI</spirit:apiType>
        <spirit:generatorExe>http://www.acme.com:1500</spirit:generatorExe>
    </spirit:componentGenerator>
```

produces the following output.

```
http://www.acme.com:1500?url=http%3a%2f%2fhost%3aport&param1=default1
    &param2=fixedValue
```

G.3 TGI API

The TGI API defines the set of legal SOAP messages that can be sent from a generator to a DE, along with the format of the responses the generator can expect from a given request (message) to the DE. The API shall provide the means of getting and setting values within the IP-XACT design currently represented in the DE. The API commands can be classified as shown in Table G.1.

Table G.1—TGI API classifications

Category	Description	Example
Get	Commands which get attribute or element values. These commands are available for getting all information from the design and component schemas. If the attribute or element does not exist, this may return a default value, an empty string, or an empty array.	Get port width.
Set	Commands which set element values. These commands are available to set each element for which the **resolve** attribute is legal. Setting the value of the element fails unless the resolve value is **user** or **generator**. Set routines return a Boolean value where a *true* return code implies a successful operation. If **false** is returned, the SOAP fault code shall provide additional information detailing the failure.	Get parameter value.
Traversal	Commands that return a list of elements, which can then be traversed for further manipulation.	Get components in a design.
Administrative	Commands that do not deal directly with the IP-XACT meta-data.	Terminate communication.

The complete set of API commands is defined using WSDL so that it can be defined in a language-independent format.

G.3.1 TGI fault codes

The fault codes for TGI failures are as follows:

1 - Unknown (undefined) error

2 - Illegal element ID

3 - Illegal value(s)

4 - Element is not modifiable (incompatible **resolve** value)

5 - Operation not supported by the DE

6 - Operation not supported in this version of the schema

7 - Operation failed

G.3.2 Administrative commands

There are three administrative commands defined in the API.

a) **Init** is the required first message from the generator to the DE. It tells the DE that the generator has properly connected via SOAP.

   1) Input

      i) **apiVersion** of type *string*—Indicates the API version for which the generator is defined to work.

      ii) **failureMode** of type ***apiFailureMode***—Compatibility failure mode

      fail indicates the DE shall return an error on the `init` call if its API version does not match the one passed to the `init` call;

      error indicates the DE shall return an error each time a potentially incompatible API call is made;

      warning indicates the DE shall increment a warning count each time a potentially incompatible API call is made.

      iii) **message** of type *string*—Message that the DE may display to the user.

   2) Returns: **status** of type ***boolean***.

b) **End** is the required last message from the generator to the DE. It tells the DE it is okay to stop listening for messages from the generator. This includes a generator return status, although the generator is not strictly required to terminate after sending the message.

c) **Message** indicates some form of generator status to pass to the user.

G.4 IDs and configurable values

Most TGI calls take an element identifier that acts as a handle or pointer to the element and are referred to as *ID*s. These IDs allow a single TGI command to operate on many different ID types to produce a result. One such example is **getDescription(*ID*)**, which takes any ID type as input and returns its description if that ID contains a description element. When an ID is passed to a TGI routine that returns an element's value, the configured value (component) is always returned. If the unconfigured value (design) is desired (the default), **getUnconfiguredID** can be used to translate the ID into an unconfigured identifier, which is referred to as a *UID*. The configured and unconfigured values may be the same. The only time the values are different is when the unconfigured (default) value is overridden via a **configureElementValue** from the design file.

The TGI API presumes the data stored in the design description to configure an element that has a **resolve** attribute value of **user** or **generated** is applied to the component instance by the DE. This enables the TGI author to simply ask for the value of an element on a given component instance and retrieve the correct answer. The setting of an element works similarly. When an element is set on a component instance, the value of this element is ultimately stored in the design description.

The design configuration description is handled in the same manner as the configurable elements as in the design description. The settings in the design configuration description are applied to the elements in the referenced design description or the containing component instances. Therefore, there are no TGI functions to retrieve the design configuration information directly; the TGI author can find this information applied to the correct element in the design or component instance. For example, the configured view of a component instance is accessed using the normal **getComponentViewIDs** with a Boolean argument set to indicate the configured view (specified in the design configuration description) should be returned.

IDs returned by TGI commands are guaranteed to be persistent for the duration of a single generator invocation provided the element being referenced is not removed. For example, if an ID represents an address space element, that ID can be utilized as often as is needed during a single generator invocation, unless the component containing the address map is removed by calling **removeComponentInstance()**.

G.5 TGI messages

The TGI is a set of messages used to query and modify an IP-XACT compliant database. The TGI messages are composed of a SOAP envelope and a TGI body. The TGI services are specified in the `TGI.wsdl` file. Each TGI body message is an XML element whose name is the name of the TGI command and whose elements are the arguments of the TGI command. All TGI messages apply to IP-XACT XML elements, identified by an ID, i.e., a TGI server-defined constant uniquely identifying an IP-XACT XML element throughout a TGI server session.

G.6 Vendor attributes

One case of special interest to a user may be the location of vendor attributes in the schema. These attributes are allowed in more places in the schema than the TGI allows a user to retrieve them. This goes back to the concept where one function uses many different ID types to return some data. In the case of vendor attributes, these can only be accessed if the containing element has an ID.

G.7 TGI SOAP messages

G.7.1 TGI SOAP message index

Abstraction definition operations

- getAbstractionDefBusTypeVLNV - Get VLNV of the bus definition.
- getAbstractionDefExtends - VLNV of the abstraction definition being extended.
- getAbstractionDefID - ID for the abstraction definition with the given VLNV.
- getAbstractionDefPortDefaultValue - Default value for port when not connected.
- getAbstractionDefPortDriveConstraintIDs - List of drive constraint IDs of the port.
- getAbstractionDefPortIDs - List of abstraction definition port element IDs.
- getAbstractionDefPortIsAddress - Is this port an address port.

- getAbstractionDefPortIsClock - Is this port a clock port.
- getAbstractionDefPortIsData - Is this port a data port.
- getAbstractionDefPortIsReset - Is this port a reset port.
- getAbstractionDefPortLoadConstraintIDs - List of load constraint IDs of the port.
- getAbstractionDefPortLogicalName - Logical name of this abstraction definition port.
- getAbstractionDefPortMirroredConstraintIDs - List of constraint IDs for a mirrored port.
- getAbstractionDefPortModeBitWidth - Bit width constraint when present on an interface of the given type.
- getAbstractionDefPortModeDirection - Port direction constraint when present on an interface of the given type.
- getAbstractionDefPortModeGroup - Group name when present on a system interface.
- getAbstractionDefPortModeIDs - Returns an array of IDs for accessing the given port in the given interface mode. The array shall only contain one element if the modeValue input is master or slave. The array may contain multiple elements for modeValue system.
- getAbstractionDefPortModePresence - Existence requirement for this port on an interface of the given type.
- getAbstractionDefPortModeServiceID - AbstractionDef service ID on a transactional port.
- getAbstractionDefPortNonMirroredConstraintIDs - List of constraint IDs for a non-mirrored port.
- getAbstractionDefPortRequiredDriverType - Required driver type for this port.
- getAbstractionDefPortRequiresDriver - Does this port require a driver.
- getAbstractionDefPortStyle - Returns wire or transactional to indicate the port style.
- getAbstractionDefPortTimingConstraintIDs - List of timing constraint IDs of the port.
- getAbstractionDefVLNV - VLNV of the abstraction definition.

Abstractor instance operations

- getAbstractorInstanceAbstractorID - ID for the abstractor associated with given instance (crossing from design configuration to abstractor file).
- getAbstractorInstanceName - Instance name of the abstractor.
- getAbstractorInstanceVLNV - VLNV of the abstractor (from the design file).
- getAbstractorInstanceXML - Return the abstractor XML in text format. Schema version is DE dependent.

Abstractor operations

- getAbstractorAbstractorInterfaceIDs - List of two interface IDs.
- getAbstractorAbstractorMode - Get the mode that the abstractor can be master, slave, direct, or system.
- getAbstractorBusTypeVLNV - List of VLNV of the bus definition.
- getAbstractorChoiceIDs - List of choices IDs.
- getAbstractorFileSetIDs - List of file set IDs.
- getAbstractorGeneratorIDs - List of generator IDs of the abstractor.
- getAbstractorModelParameterIDs - A list of model parameter IDs.
- getAbstractorPortIDs - A list of abstractor model port IDs.
- getAbstractorViewIDs - A list of model view IDs.

Address map operations

- getAddressBlockAccess - The accessibility of the data in the local address block.
- getAddressBlockBaseAddress - The base address of an address block.

- getAddressBlockRange - The address range of an address block expressed as the number of accessible and addressable units.
- getAddressBlockRegisterFileIDs - The IDs of the available register files in the address block.
- getAddressBlockRegisterIDs - The IDs of the available registers in the address block.
- getAddressBlockUsage - Indicates the usage of this address block.
- getAddressBlockVolatility - Indicates whether or not the data is volatile.
- getAddressBlockWidth - The bit width of an address block in the local memory map.
- getAddressSpaceAddressUnitBits - The number bits in an addressable unit. If none exists, the default 8 bits is returned.
- getAddressSpaceLocalMemoryMapID - The ID for the local memory map of the address space.
- getAddressSpaceRange - The address range of an address block expressed as the number of accessable and addressable units.
- getAddressSpaceSegmentIDs - List of IDs for address block segments for the address space.
- getAddressSpaceWidth - The bit width of an address block.
- getBankAccess - The accessibility of the data in the local address bank.
- getBankAlignment - The bank alignment value, serial or parallel.
- getBankBaseAddress - The base address of an address bank.
- getBankUsage - Indicates the usage of this address bank.
- getBankVolatility - Indicates whether or not the data is volatile.
- getExecutableImageFileBuilderIDs - List of default file builder IDs of the executable image.
- getExecutableImageFileSetIDs - The group of file set reference IDs complying with the tool set of the current executable image.
- getExecutableImageIDs - The IDs of the executable images belonging to the specified address space.
- getExecutableImageLinkerCommand - The linker command for the current executable image.
- getExecutableImageLinkerCommandFileID - Element ID of linkerCommandFile associated with given executable image.
- getExecutableImageLinkerFlags - The flags of the current executable image linker command.
- getExecutableImageType - The type of the executable image if existent.
- getLinkerCommandFileEnable - Indicates whether or not to generate and enable the linker command file.
- getLinkerCommandFileLineSwitch - The command line switch to specify with the linker command file.
- getLinkerCommandFileName - The name of the linker command file.
- getLinkerCommandGeneratorIDs - Reference IDs to the generator elements for generating the linker command file.
- getMemoryMapAddressUnitBits - The number bits in an addressable unit for a memory map. If none exists, the default 8 bits is returned.
- getMemoryMapElementIDs - List of element IDs (addressBlockID, bankID, subspaceMapID) within a memory map, memory remap, local memory map, or bank.
- getMemoryMapElementType - Indicates type of memory map element: addressBlock, bank, or subspaceMap.
- getMemoryMapRemapElementIDs - List of IDs for memory map remap elements of the given memory map.
- getMemoryRemapStateID - Remap State ID for which this remap is applicable.
- getSegmentAddressOffset - The address offset of an address space segment in an address space.

- getSegmentRange - The address range of an address space segment expressed as the number of accessable addressable units.
- getSubspaceMapBaseAddress - The base address of a memory subspace.
- getSubspaceMapMasterID - Master bus interface ID on the other side of a bus bridge.
- getSubspaceMapSegmentID - Address space segment ID on the other side of a bus bridge.
- getTypeIdentifier - Indicates the type identifier of an addressBlock, registerFile, register, or field.
- setAddressBlockBaseAddress - Set the base address of an address block.
- setAddressBlockRange - Set the address range of an address block expressed as the number of accessable and addressable units.
- setAddressBlockWidth - Set the bit width of an address block.
- setAddressSpaceRange - Set the address range of an address block expressed as the number of accessable and addressable units.
- setAddressSpaceWidth - Set the bit width of an address block.
- setBankBaseAddress - Set the base address of an address bank.
- setExecutableImageLinkerCommand - Set the linker command for the current executable image.
- setExecutableImageLinkerFlags - Set the flags of the current executable image linker command.
- setLinkerCommandFileEnable - Set whether or not to generate and enable the linker command file.
- setLinkerCommandFileLineSwitch - Set the command line switch to specify with the linker command file.
- setLinkerCommandFileName - Set the name of the linker command file.
- setSegmentAddressOffset - Set the address offset of an address space segment expressed in the number addressable units.
- setSegmentRange - Set the address range of an address space segment expressed as the number of accessable addressable units.
- setSubspaceMapBaseAddress - Set the base address of a memory subspace.

Bus definition operations

- getBusDefinitionDirectConnection - Indicates whether or not the bus definition supports direct connections.
- getBusDefinitionExtends - VLNV of the bus definition being extended.
- getBusDefinitionID - ID for the bus definition with the given VLNV.
- getBusDefinitionIsAddressable - Indicates whether or not the bus definition is an addressable bus.
- getBusDefinitionMaxMasters - Maximum # of masters supported by this bus definition.
- getBusDefinitionMaxSlaves - Maximum # of slaves supported by this bus definition.
- getBusDefinitionSystemGroupNames - List of system group names for this bus definition.
- getBusDefinitionVLNV - VLNV of the bus definition.

Bus interface operations

- getBridgeIsOpaque - Value of the opaque attribute.
- getBridgeMasterID - The slave interface or master interface reference ID.
- getBusInterfaceBitSteering - Bit steering description of the bus interface: on or off.
- getBusInterfaceBitsInLAU - The number bits in the least addressable unit. If none exists, the default 8 bits is returned.
- getBusInterfaceConnectionRequired - Connection required for this bus interface.
- getBusInterfaceEndianness - The endianess of the bus interface, big or little. The default is little.
- getBusInterfaceGroupName - Group name of a system, mirroredSystem, or monitor bus interface.
- getBusInterfaceMasterAddressSpaceID - ID of the master addressSpace.

- getBusInterfaceMasterBaseAddress - Base address of the master addressSpace.
- getBusInterfaceMirroredSlaveRange - The address range of the mirrored slave interface.
- getBusInterfaceMirroredSlaveRemapAddressIDs - List of remap address IDs of the mirrored slave interface.
- getBusInterfaceMonitorInterfaceMode - Indicates the mode of interface being monitored, slave, master, system, mirrorslave, mirrormaster, or mirrorslave.
- getBusInterfaceSlaveBridgeIDs - List of slave bridge IDs.
- getBusInterfaceSlaveFileSetGroupIDs - List of fileSetGroup IDs.
- getBusInterfaceSlaveMemoryMapID - ID of the memoryMap referenced from a slave interface.
- getRemapAddressRemapStateID - Remap state ID of the given remap address element.
- getRemapAddressValue - Remap address of the given remap address element.
- setBusInterfaceBitSteering - Set bus interface bit steering value.
- setBusInterfaceMasterBaseAddress - Set base address of the master bus interface.
- setBusInterfaceMirroredSlaveRange - Set address range for the associated interface.
- setRemapAddressValue - Set remap address value for the associated interface.

Component instance operations

- getComponentInstanceComponentID - ID for the component associated with given instance (crossing from design to component file).
- getComponentInstanceName - Instance name of the component.
- getComponentInstanceVLNV - VLNV of the component (from the design file).
- getComponentInstanceXML - Return the component XML in text format. Schema version is DE dependent.

Component operations

- getChannelBusInterfaceIDs - List of busInterface IDs in this channel.
- getComponentAddressSpaceIDs - List of IDs for the logical address spaces in the component.
- getComponentBusInterfaceIDs - List of interface IDs.
- getComponentChannelIDs - A list of channel IDs.
- getComponentChoiceIDs - List of choices IDs.
- getComponentCpuIDs - List of cpu IDs of the component.
- getComponentElementType - Returns the type name of the XML element associated with the component (currently only component). This call is being provided to cover a future scenario where there can be different types of component elements instantiated in a design (e.g., macroComponent elements).
- getComponentFileSetIDs - List of file set IDs.
- getComponentGeneratorIDs - List of generator IDs of the component.
- getComponentMemoryMapIDs - List of IDs for memory map elements in the given component.
- getComponentModelParameterIDs - A list of model parameter IDs.
- getComponentOtherClockDriverIDs - List of clock driver IDs of the component.
- getComponentPortIDs - A list of component model port IDs.
- getComponentRemapStateIDs - A list of remap state IDs.
- getComponentVLNV - VLNV of the component (from the component file).
- getComponentViewIDs - A list of model view IDs.
- getComponentWhiteboxElementIDs - List of white box element IDs of the component.
- getCpuAddressSpaceIDs - List of address space reference IDs of the cpu.

Constraint operations

- getDriveConstraintType - Indicates the type of drive constraint: function class.
- getDriveConstraintValue - Returns the drive constraint. Format depends on the constraint type.
- getLoadConstraintCount - Returns the load constraint count, the number of loads.
- getLoadConstraintType - Indicates the type of load constraint: function class.
- getLoadConstraintValue - Returns the load constraint. Format is cell function and strength or cell class and strength.
- getPortConstraintSetDriveConstraintIDs - List of drive constraint IDs of the port.
- getPortConstraintSetLoadConstraintIDs - List of load constraint IDs of the port.
- getPortConstraintSetRange - List of the left and right range of a port referenced by this constraint set.
- getPortConstraintSetReferenceName - Reference name of the given port constraint set.
- getPortConstraintSetTimingConstraintIDs - List of timing constraint IDs of the port.
- getTimingConstraintClockDetails - Indicates the clock name, clock edge, and delay type.
- getTimingConstraintValue - Returns the timing constraint value (cycle time percentage).

Design operations

- addAdHocConnection - Add new ad hoc connection.
- addAdHocExternalPortReference - Add an external port reference to an existing ad hoc connection.
- addAdHocInternalPortReference - Add an internal port reference to an existing ad hoc connection. An identical port reference must not already exist in the ad hoc connection.
- addComponentInstance - Add new component instance.
- addHierConnection - Add new hierarchical connection.
- addHierarchicalMonitorInterconnection - Add new hierarchical interconnection between a component and monitor. If there is already a monitorInterconnection for the given componentRef/componentInterfaceRef, then the monitor connection is added to that element.
- addInterconnection - Add new interconnection between components.
- addMonitorInterconnection - Add new interconnection between a component and monitor. If there is already a monitorInterconnection for the given componentRef/componentInterfaceRef, then the monitor connection is added to that element.
- appendAbstractorInstance - Append a new abstractor instance to the interconnection.
- getAdHocConnectionExternalPortDetails - List for an external connection containing the portRef, left, and right attribute values.
- getAdHocConnectionExternalPortReferenceIDs - List of external ad hoc port reference element IDs.
- getAdHocConnectionInternalPortReferenceDetails - List for an internal connection containing the componentRef, portRef, left, and right attribute values.
- getAdHocConnectionInternalPortReferenceIDs - List of internal ad hoc port reference element IDs.
- getAdHocConnectionTiedValue - Get the tied value for an ad hoc connection.
- getComponentInstanceID - Return the component instance ID of the named component instance in the given design.
- getDesignAdHocConnectionIDs - List of ad hoc connection element IDs.
- getDesignComponentInstanceIDs - Components instances IDs of the given design.
- getDesignHierConnectionIDs - List of hierarchical connection element IDs.
- getDesignID - Get ID of the current or top design.
- getDesignInterconnectionAbstractorInstanceIDs - List of abstractor instances IDs for this interconnection.
- getDesignInterconnectionIDs - List of interconnection element IDs.

- getDesignMonitorInterconnectionIDs - List of monitorInterconnection element IDs.
- getDesignVLNV - VLNV of the design.
- getHierConnectionDetails - List containing the interface name, component reference, and interface reference.
- getInterconnectionActiveInterfaces - Returns the active interfaces as a list: componentRef interfaceRef componentRef interfaceRef.
- getMonitorInterconnectionInterfaces - Returns the active interface and monitor interfaces as a list in componentPathRef, componentRef, componentInterface, monitorPathRef, monitorRef, monitorInterface format; the active interface comes first in the list.
- removeAbstractorInstance - Remove specified abstractor instance.
- removeAdHocExternalPortReference - Remove an external port reference from existing ad hoc connection.
- removeAdHocInternalPortReference - Remove an internal port from existing ad hoc connection. The ad hoc connection is removed when the last port reference is removed.
- removeComponentInstance - Remove specified component instance.
- removeHierConnection - Remove existing hierarchical connection.
- removeHierarchicalMonitorInterconnection - Remove a hierarchical interconnection between a component and monitor. When the last monitor reference is removed, the entire monitorInterconnection element will be removed.
- removeInterconnection - Remove interconnection between components, and any abstractors if present.
- removeMonitorInterconnection - Remove interconnection between a component and monitor. When the last monitor reference is removed, the entire monitorInterconnection element will be removed.
- replaceAbstractorInstance - Replace specified abstractor with new provided abstractor.
- replaceComponentInstance - Replace specified component with new provided component.

Field operations

- getRegisterFieldAccess - The accessibility of the data in the field.
- getRegisterFieldBitOffset - Bit offset of the fields LSB inside the register.
- getRegisterFieldBitWidth - Width of the field in bits.
- getRegisterFieldModifiedWriteValue - The modified write value for the field.
- getRegisterFieldReadAction - The read action for the field.
- getRegisterFieldTestConstraint - The test constraint required if the field can be tested with a simple register test.
- getRegisterFieldTestable - True if the field can be tested with a simple register test.
- getRegisterFieldValue - Enumerated bit field value.
- getRegisterFieldValueIDs - List of IDs for field values for the given register field.
- getRegisterFieldValueName - Enumerated name for this register field value. Deprecated—use getName.
- getRegisterFieldValueUsage - Enumerated bit field usage.
- getRegisterFieldVolatility - Indicates whether or not the data is volatile. The presumed value is false if the element is not present.
- getRegisterFieldWriteValueConstraintMinMax - The value of a write constraint.
- getRegisterFieldWriteValueConstraintUseEnumeratedValues - The write value constraint shall use the enumerated values.
- getRegisterFieldWriteValueConstraintWriteAsRead - The write value constraint is write as read.
- setRegisterFieldBitWidth - Set the width of the field in bits.

File and fileset operations

- getFileBuildCommandFlags - Flags of the file build command.
- getFileBuildCommandFlagsIsAppend - Value of append attribute on the flag element.
- getFileBuildCommandName - Name of the build command of the file.
- getFileBuildCommandReplaceDefaultFlags - Indicates whether or not to replace default flags.
- getFileBuildCommandTargetName - Target name of the file build command.
- getFileBuilderCommand - Command of the file builder.
- getFileBuilderFileType - FileType or userFileType of the file builder.
- getFileBuilderFlags - Flags of the file builder.
- getFileBuilderReplaceDefaultFlags - Value of the replaceDefaultFlags element of the file builder.
- getFileDefineSymbolIDs - List of define symbol IDs used in the file.
- getFileDependencies - List of dependent locations for the file, typically directories.
- getFileExportedNames - List of exported names of the file.
- getFileHasExternalDeclarations - Indicates that the file includes external declarations required by the top-level netlist file.
- getFileImageTypes - List of image types of the file.
- getFileIsIncludeFile - Indicates that the given file is an include file.
- getFileLogicalName - Logical name of the file.
- getFileLogicalNameDefault - Default attribute of logical name of the file.
- getFileName - Get name of the given fileID.
- getFileSetDependencies - List of dependent locations for the fileSet, typically directories.
- getFileSetFileBuilderIDs - List of file builder IDs used for this fileSet.
- getFileSetFileIDs - List of file IDs of the fileSet.
- getFileSetFunctionIDs - List of function IDs.
- getFileSetGroupFileSetIDs - List of fileSet IDs in this file set group.
- getFileSetGroupName - Name of fileSet group.
- getFileSetGroups - List of group names of the fileSet.
- getFileType - FileType or userFileType of the file.
- getFunctionArgumentDataType - Data type of the argument.
- getFunctionArgumentIDs - List of argument IDs of the function of the fileSet.
- getFunctionDisabled - Indicates whether or not the function is disabled.
- getFunctionEntryPoint - Entry point of the function.
- getFunctionFileID - File ID containing the function entry point.
- getFunctionReplicate - Value of replicate attribute on function element.
- getFunctionReturnType - Return type of the function.
- getFunctionSourceFileIDs - List of source file IDs of the function of the fileSet.
- getFunctionSourceFileName - Name of the source file.
- getFunctionSourceFileType - FileType or userFileType of the source file.
- setFileBuildCommandFlags - Set command flags for the given file builder.
- setFileBuildCommandName - Set command name for the given file builder.
- setFileBuildCommandReplaceDefaultFlags - Set replace default flags for the given file builder.
- setFileBuildCommandTargetName - Set target name for build command for the given file.
- setFileBuilderCommand - Set command associated with file builder.

- setFileBuilderFlags - Set flags associated with the given file builder.
- setFileBuilderReplaceDefaultFlags - Set value of replace default flags in file builder.
- setFileName - Set name of the given file.
- setFunctionDisabled - Set disable flag on function.

Generator operations

- getGeneratorApiType - Api type of the generator.
- getGeneratorExecutable - Executable name associated with the generator.
- getGeneratorGroups - List of group names of the generator.
- getGeneratorIsHidden - Value of hidden attribute on the generator.
- getGeneratorPhase - Phase number of the generator.
- getGeneratorScope - Scope of the generator.
- getGeneratorTransportMethods - List of transport methods of the generator.

Interface operations

- getInterfaceAbstractionTypeVLNV - List of VLNV of the abstraction definition.
- getInterfaceBusTypeVLNV - List of VLNV of the bus definition.
- getInterfaceMode - Mode of the interface: master, slave, system, mirroredMaster, mirroredSlave, mirroredSystem, or monitor.
- getInterfacePortMapIDs - List of interface port map IDs.
- getLogicalPhysicalMapIDs - List of the logical and physical port map IDs.
- getPortMapRange - List of left and right range of the port map.
- setPortMapRange - Set left/right range of an interface port map.

Miscellaneous operations

- end - Terminate connection to the DE.
- getChoiceEnumerationHelp - Value of the enumeration help attribute.
- getChoiceEnumerationIDs - List of choice enumeration IDs of the choice.
- getChoiceEnumerationText - Value of the enumeration text attribute.
- getChoiceEnumerationValue - Value of the enumeration element.
- getChoiceName - Name of the choice.
- getDescription - Return the description of the specified element.
- getDisplayName - Return the displayName of the specified element.
- getErrorMessage - Get error message from prior callback.
- getGeneratorContextComponentInstanceID - ID for the component instance associated with the currently invoked generator.
- getIdValue - Return the value of the spirit:id attribute on a ID.
- getModelParameterDataType - Data type of the model parameter.
- getModelParameterUsageType - Usage type of the model parameter.
- getName - Return the name of the specified element.
- getParameterIDs - List of parameter IDs from the given element (any of which contains spirit:parameter elements).
- getUnconfiguredID - Return the unconfigured ID from a configured ID.
- getValue - Get the value of a parameterID, fileDefineIDs, or argumentIDs.
- getValueAttribute - Returns the value of the given attribute name on the elementID/value element.
- getVendorAttribute - Get vendor-defined attribute from the given element.

- getVendorExtensions - Returns the complete XML text of the vendor extension element including the spirit:vendorExtension tag, as a well-formed XML document.
- getWarningCount - Return count of how many potentially incompatible API calls have been made.
- getXMLForVLNV - Return XML of the IP-XACT object identified by the given VLNV.
- init - API initialization function. Must be called before any other API call.
- message - Send message level and message text to DE.
- registerVLNV - Indicate to DE where the file resides for the IP-XACT element with the given VLNV.
- setValue - Set the value of a parameterID, fileDefineIDs, or argumentIDs.
- setVendorAttribute - Set vendor-defined attribute on the given element.
- setVendorExtensions - Set vendor extensions. See NOTE.

Port operations

- getAllLogicalDirectionsAllowed - Get the value of the allLogicalDirectionAllowed attribute.
- getClockDriverName - Name of the clock driver.
- getClockDriverPeriod - Clock period of the given clock.
- getClockDriverPeriodUnits - Units of the clock period of the given clock.
- getClockDriverPulseDuration - Clock period of the given clock.
- getClockDriverPulseDurationUnits - Units of the clock pulse duration of the given clock.
- getClockDriverPulseOffset - Clock pulse offset of the given clock.
- getClockDriverPulseOffsetUnits - Units of the clock pulse offset of the given clock.
- getClockDriverPulseValue - Clock pulse value of the given clock.
- getClockDriverSource - Source name of the clock driver.
- getPortAccessHandle - Alternate name to be used when accessing this port.
- getPortAccessType - Indicates the access type for this port.
- getPortClockDriverID - Element ID of clock driver element, if present.
- getPortConstraintSetIDs - List of constraint sets IDs of the port.
- getPortDefaultValue - Default value of the port, if not set returns "".
- getPortDirection - Direction of the port.
- getPortMaxAllowedConnections - Maximum allowed connections for this transactional port.
- getPortMinAllowedConnections - Minimum allowed connections for this transactional port.
- getPortRange - List of the left and right range of the port.
- getPortServiceID - ID of element representing the service of a transactional port.
- getPortSingleShotDriverID - Element ID of single shot driver element, if present.
- getPortSingleShotPulseDuration - Clock period of the port.
- getPortSingleShotPulseOffset - Clock pulse offset of the port.
- getPortSingleShotPulseValue - Clock pulse value of the port.
- getPortStyle - Returns wire or transactional to indicate the port style.
- getPortTransactionalTypeDefID - The type definition for a transactional portID.
- getPortWireTypeDefIDs - List of typeDefs for a wire portID.
- setClockDriverPeriod - Set period of the given clock port.
- setClockDriverPulseDuration - Set pulse duration of the given clock port.
- setClockDriverPulseOffset - Set pulse offset value of the given clock port.
- setClockDriverPulseValue - Set pulse value of the given clock port.

- setPortDefaultValue - Set default value of the given port.
- setPortRange - Set left/right range for the given port.
- setPortSingleShotPulseDuration - Set pulse duration of given single shot port.
- setPortSingleShotPulseOffset - Set pulse offset of given single shot port.
- setPortSingleShotPulseValue - Set pulse value of given single shot port.

Register file operations

- getRegisterFileAddressOffset - The offset from the base address.
- getRegisterFileDimensions - Dimensions of a register file array.
- getRegisterFileRange - The register file range in number of addressable units.
- getRegisterFileRegisterFileIDs - List of IDs for the register files of the given register file.
- getRegisterFileRegisterIDs - List of IDs for the registers of the given register file.
- setRegisterFileRange - Set the register file range in addressable units.

Register operations

- getRegisterAccess - The accessibility of the data in the register.
- getRegisterAddressOffset - The offset from the base address.
- getRegisterAlternateGroups - Indicates the group names for an alternate register.
- getRegisterAlternateRegisterIDs - List of IDs for the alternate registers of the given register.
- getRegisterDimensions - Dimensions of a register array.
- getRegisterFieldIDs - List of IDs for the fields of the given register.
- getRegisterResetMask - Mask to be ANDed with the value before comparing to reset value.
- getRegisterResetValue - Register value at reset.
- getRegisterSize - The register size in bits.
- getRegisterVolatility - Indicates whether or not the data is volatile.
- setRegisterResetMask - Set the mask to be ANDed with the value before comparing to reset value.
- setRegisterResetValue - Set register value at reset.
- setRegisterSize - Set the register size in bits.

Remap operations

- getRemapStatePortIDs - List of remap port IDs of a remap state.
- getRemapStatePortPortID - Port ID for the remap state.
- getRemapStatePortPortIndex - Index of the port if a vector for the remap state.
- getRemapStatePortPortValue - Value of the port for the remap state.

Service operations

- getAbstractionDefAbstractionServiceTypeDefIDs - List of type definitions for an abstractionServiceID.
- getAbstractionDefServiceInitiative - Port service initiative from the abstraction definition.
- getServiceInitiative - Initiative of the service.
- getServiceTypeDefIDs - List of typeDefs for a serviceID.

Typedef operations

- getTypeDefConstrained - Is the type name constrained?
- getTypeDefImplicit - Is the type name implicit?
- getTypeDefTypeDefinitions - List of type definition for the given type.
- getTypeDefTypeName - Name of the type.
- getTypeDefTypeViewIDs - List of type viewIDs for the given type.

View operations

- getViewDefaultFileBuilderIDs - List of default file builder IDs of the view.
- getViewDesignID - ID of the design associated with a hierarchical view.
- getViewEnvIdentifiers - List of environment identifiers of the view.
- getViewFileSetIDs - List of fileSet IDs for fileSets referenced by the view.
- getViewLanguage - View Language.
- getViewLanguageIsStrict - Value of strict attribute on view language element.
- getViewModelName - Get the model name for this view.
- getViewPortConstraintSetIDs - Constraint set ID for the port referenced by the view.
- getViewWhiteboxElementRefIDs - List of white box element reference IDs of the view.

White box operations

- getWhiteboxElementDrivable - Indicates whether or not the white box element is drivable.
- getWhiteboxElementRefID - White box element reference ID.
- getWhiteboxElementRegisterIDs - Register reference IDs of the white box element.
- getWhiteboxElementType - Type of the white box element.
- getWhiteboxRefPathIDs - List of path IDs of the white box element reference.
- getWhiteboxRefPathName - Name of the white box reference path element.
- getWhiteboxRefPathRange - List of left and right range of the white box reference path element.

G.7.2 Abstraction definition operations

G.7.2.1 getAbstractionDefBusTypeVLNV

Description: Get VLNV of the bus definition.

- Input: `abstractionDefID` of type ***xsd:string***.
- Returns: `vlnvValue` of type ***spirit:soapStringArrayType***.

G.7.2.2 getAbstractionDefExtends

Description: VLNV of the abstraction definition being extended.

- Input: `abstractionDefID` of type ***xsd:string***.
- Returns: `vlnvValue` of type ***spirit:soapStringArrayType***.

G.7.2.3 getAbstractionDefID

Description: ID for the abstraction definition with the given VLNV.

- Input: `vlnvValue` of type ***spirit:soapStringArrayType***.
- Returns: `abstractionDefID` of type ***xsd:string***.

G.7.2.4 getAbstractionDefPortDefaultValue

Description: Default value for port when not connected.

- Input: `abstractionDefPortID` of type ***xsd:string***.
- Returns: `value` of type ***xsd:string***.