ANSYS

IP Design Exchange Format for FMEDA and Fault Injection (IPD-XML)

Document ID: IPD-XML-TS-001

Technical Specification

Revision: 1.2

TABLE OF CHANGE RECORDS

Revision	Date	Change record	Author(s)
1.0	2018-07-04	First public version of the IPD-XML Technical Specification. Fully compliant with the internal spec Rev. 0.5 "*FMEDA and Fault Injection Exchange Format for IP Designs*"	Michael Soden
1.1	2018-12-01	Revised version with smaller refinements for cell and gate count attributes	Michael Soden
1.2	2019-04-12	Improvements to fault injection exchange section; Major schema upgrade	Michael Soden

TABLE OF CONTENTS

LIST OF FIGURES

DOCUMENT RELEASE FORM

Document Repository: SAFETY

Document Title: IP Design Exchange Format for FMEDA and Fault Injection (IPD-XML)

Document Identifier: IPD-XML-TS-001

Document Revision: 1.2

Document Date: 2019-04-12

Document Classification: PUBLIC

Abstract

This specification describes the exchange format for performing *Failure Mode, Effects, and Diagnostic Analysis* (FMEDA) on a chip design (IP Design) with support for verification of diagnostic coverage by fault injection. The format is named IPD-XML and implemented by the ANSYS product *medini analyze*.

Author(s):	LEAD PRODUCT MANAGER Michael SODEN

Distribution List

PUBLIC (Website)

1 INTRODUCTION

1.1 PURPOSE

A key challenge for complex semiconductor chips is to perform safety and reliability assessments for application in safety and mission critical systems. Safety standards such as IEC 61508, ISO 26262, ARP4761, EN 50126, and others have identified the *Failure Mode, Effects, and Diagnostic Analysis* (FMEDA) as the state-of-the-art method to assess and quantify the failure characteristics of an IC.

With the advent of ISO/PAS 19451 (2016) and the corresponding ISO 26262, Part 11 "*Guidelines on application of ISO 26262 to semiconductors*" (2018) a detailed approach exists to perform FMEDA in the context of Automobile systems. These standards provide especially guidelines for the following key activities:

1. Definition of failure modes of a variety of ICs (digital, analog, mixed-signal, etc.)
2. Quantification of (base) failure rates and application to complex ICs
3. Distribution and allocation of failure rates across chip design and package
4. Determining failure rates using handbook data (IEC 62380, SN29500, FIDES Guide) or accelerated life testing
5. Treatment of transient failures in the analysis
6. Verification of diagnostic coverage claimed by fault injection
7. Checklists for common cause initiators

To exchange design information such as instance hierarchies, transistor/gate counts, die area, and related seamlessly between tools for functional safety, we have defined an exchange format especially for safety and reliability analysis. The data of a design at different abstraction levels (Behavior, RTL or NL) in any HDL can be mapped into this format to allow processing of the relevant data for the following two use cases:

I. *Failure rate determination and distribution*: failure rates are determined at various levels of a design (complete chip, die, parts, sub-parts, etc.) and can be distributed based on (relative) spatial figures of the instances. Depending on the technology either area, transistor, cell, or gate counts might be used for this purpose. For transient failures especially, the number of gates and/or FF are facilitated combined with base failure rates per gate/FF/bit.

II. *Diagnostic Coverage Evaluation (Fault Injection Techniques)*: The determination of (random) runtime faults and their detection is at the core of any FMEDA analysis (e.g. SPF/LF metrics of ISO 26262, SFF of IEC 61508). This specification supports the definition of *failure modes* and *safety mechanisms* in a generic manner to exchange and bind this information to the design. The exchange will allow for generation of this information by a safety engineer and automated processing by fault injection tool suites.

Note that this specification does not prescribe how data is processed but serves only as an enabler between toolsets for functional safety and reliability analysis and H/W development environments to seamlessly exchange information. This specification aims to be independent of a specific Hardware Description Language (HDL) or modelling/design languages. However, the ANSYS medini toolchain provides a dedicated FMEDA and fault injection workflow which is centered around the IPD-XML exchange and mapping most of the concept to a SysML (UML) profile. These parts are not included in this specification. Please contact the document owner(s) for further information.

1.2 GLOSSARY AND ACRONYMS

1.2.1 Safety, Reliability, and Modelling Acronyms

Acronym	Meaning
BIT	Built-In-Test

CEA	Cascading Effects Analysis
CMA	Common Mode Analysis
FM	Failure Mode
FMEA	Failure Modes and Effects Analysis
FTA	Fault Tree Analysis
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
UML	Unified Modeling Language

1.2.2 Semiconductor Acronyms

Acronym	**Meaning**
BIST	Built-In-Self-Test
FF	Flip flop
H/W	Hardware
HDL	Hardware Description Language (e.g. Verilog, VHDL, SystemC)
HIRF	High Intensity Radiated Fields
NL	Netlist
RTL	Register Transfer Level
SEU	Single-Event Upset
SET	Single-Event Transient (momentary voltage excursion/voltage spike)
S/W	Software

2 FMEDA TOOLING WORKFLOW

2.1 CONTEXT OUTLINE

This specification describes content and format for a data exchange between safety and reliability analysis tools and tools used for Hardware development and chip design. The main purpose are the use cases failure rate prediction and fault injection validation in the context of an FMEDA (SPF/LF Metric). The specification includes the definition of the *IPD-XML* exchange format as a MOF [3] metamodel and corresponding XML Schema (based on XMI [4]). The assumed workflow between the FMEDA tool and the IP Design and fault injection tool is outlined in the following picture:

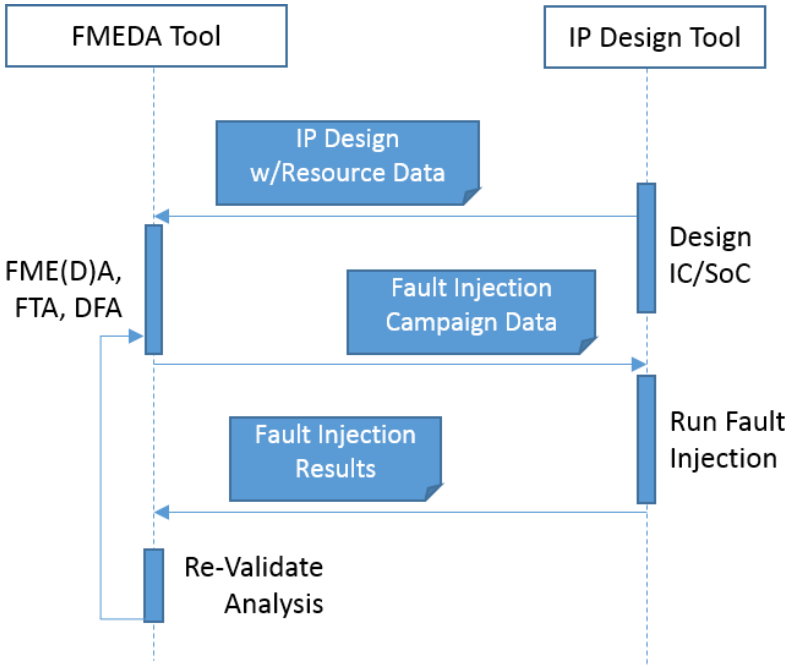

Figure 1: Relationship between Safety Tool and IP Design Tool

The IPD-XML format hierarchy information is extracted from a chip design and contains the die area of an instance, the sequential element count, and cell counts. This information will be used by an FMEDA solution such as medini analyze to compute/refine the failure-in-time (FIT) rate in FMEDA analysis and other safety analyses (FMEA, FTA, Dependent Failure Analysis, Common Cause Analysis, etc.). Based on a failure mode analysis in an FME(D)A dedicated fault injection campaigns can be initiated to validate/verify assumptions about the failures and especially diagnostic coverage (DC) of safety mechanisms.

In this specification we refer to *Fault Injection Simulation Campaigns* (FISC) as a general term to determine the diagnostic coverage (DC) of a failure mode by means of an automated analysis of a chip design. This requires a “good run” (usually by simulating the chip design), modifying the chip’s inputs by injecting faults, measuring the outputs, and performing a comparison. Any other technological approach might be applied as substitution if it leads to similar results (mathematical analysis, wafer testing, etc.)

Each *failure mode under test* (FMUT) is specified by means of an analysis (e.g. FMEA, FMEDA, FTA) and mapped to an IP design by means of *observation points*: the places in a design where the FMUT will manifest as a deviation from the specified behavior (i.e. different between good run and the run with injected fault). During a FISC, the diagnostic coverage is measured through the specification of *detection points*. These are the places where a safety mechanism flags the detection of an injected fault.

Moreover, the analysis can drive fault injection campaigns by means of *fault lists*. Fault lists specify the relevant parts of a design where faults need to be injected to provoke the failure mode under consideration.

Note that this specification does not make any assumption about confidence levels of the determined DC by fault injection nor the consistency of the mapping between failure mode, observation points, fault lists, and detection points.

2.2 LIFECYCLE

A generated XML file contains a certain snapshot of a chip design. The data schema provides a *version* tag (see section 3) to identify the design, which can be freely set by the tools during export. The FMEDA tool consuming the XML data shall preserve this version and provide it back for fault injection so that the HW tool can check or map this to a previously provided design version. By this means, the tools have a common identifier for communication of the design and analysis data.

During the exchange process, the design hierarchy is meant to be read by the FMEDA tool, but not changed. The FMEDA tool is supposed to extend the data by means of failure modes, observation/detection points, and safety mechanisms. The IP design tool can assume that the hierarchy coming back from the FMEDA is unchanged and only check for the version identifier. However, both sides are free to implement additional checksums such as SHA-1 keys over certain parts of the content to prevent user mistakes and errors.

3 METAMODEL AND XML SCHEMA

3.1 IPD-XML File Format

The IPD-XML format is defined by a metamodel using the Eclipse Modelling Framework (EMF) as metamodeling technology, a technical implementation of the Meta-Object Facility (MOF [3]). The XML schema is the corresponding XMI schema derived from the metamodel (see [4]). For convenience, the XML schema is provided in section 4.

The metamodel contains mainly two parts:

1. A design hierarchy that contains the essential data of an IP Design as unified representation of various HDL formats (RTL, VHDL, Verilog, netlists, SystemC, etc.). Only the information relevant to a safety analysis is conveyed with the format, not the full design logic. At the current stage, there is no connectivity information conveyed by this exchange format.
2. Fault Injection Data to exchange information about failure modes, safety mechanisms with corresponding references into the hierarchy to specify observation points, detection points, fault lists and DCs coming back from fault injection

The schema uses the following namespace properties:

Schema URI: http://www.ikv.de/medini/ipdesign/1.2

XMI: http://www.omg.org/XMI

XSI: http://www.w3.org/2001/XMLSchema-instance

3.2 Design Hierarchy Data

This specification refers in general to *instances of the IP design*, which means the hierarchy of objects and packaging concepts in a HW chip design. The term is meant to cover especially the following concepts:

1. Verilog module instances, hierarchical instances, objects, class instances, etc. Cell instances are instances instantiated by cell modules which are tagged with directive `celldefine` and `endcelldefine`.
2. VHDL blocks, entities, component instances, etc.
3. Netlists elements, sub-circuits, etc.
4. IP-XACT component instances, hierarchical components, design instances, etc.

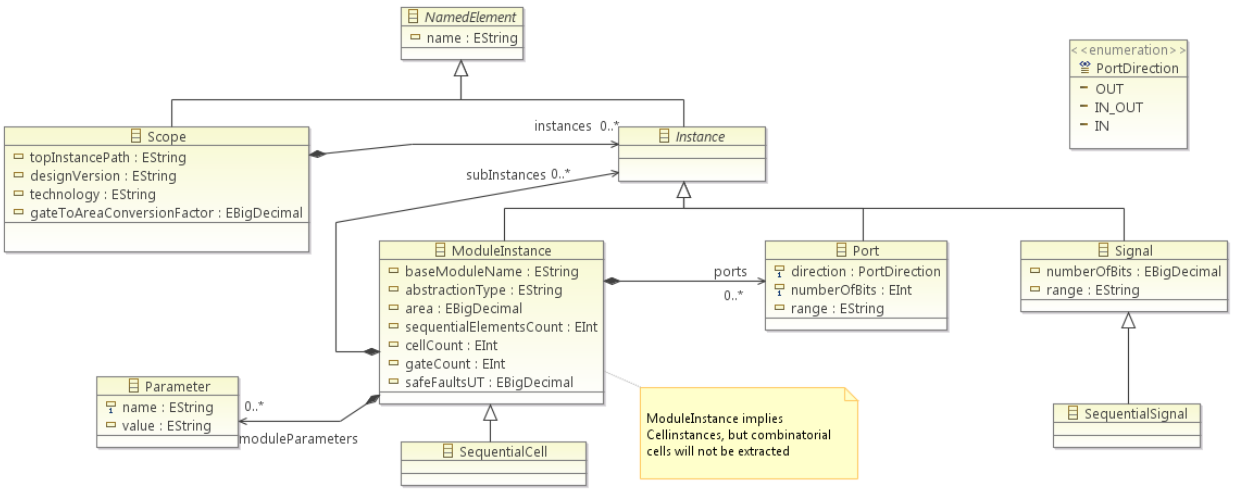

Figure 2. Metamodel for design hierarchy data (graphical representation of the XML Schema)

The metamodel classes in Figure 2 define the instance hierarchy and all attributes. The classes are described in sections 3.2.2 to 3.2.7.

3.2.1 Class *NamedElement*

Class *NamedElement* serves as abstract base class for common attributes.

Attributes

1. Attribute `name` shall be unique at each level of elements (siblings) in the containment hierarchy defined by sub-classes.
2. Note: the XMI schema provides in addition a generic ID attribute `xsi:id` which is optional and can be used as a unique identifier for elements.

3.2.2 Class *Scope*

The top-level class “Scope” defines the root object for the exchange data and contains all extracted design hierarchies.

Attributes

1. Attribute `topInstancePath` indicates the path to the top instance
2. Attribute `designVersion` is used to ensure that the design used for FMEDA analysis is the same design for functional safety analysis. It is set to “undefined” and needs to be replaced with real version number by user after the XML file has been generated.
3. Attribute `technology` describes the chip technology chosen to implement the design and that was used as basis for the die area determination
4. Attribute `gateToAreaConversionFactor` specifies the factor to be used to convert the gate count of instances into a corresponding die area (optional)
5. A set of top-level `instances` contained (usually only a single root). Note that Attribute `xsi:type` is used in XML to indicate the concrete type of the element, i.e. `ModuleInstance`, `SequentialCell` or a `Signal`.

3.2.3 Class *Instance*

Abstract base-class for all types of instances that can be contained in the extracted design hierarchy

Attributes

1. Attribute `name` (inherited from `NamedElement`) the instance name (not complete path). The name must be unique under all siblings at the same level.
2. Zero or more `subInstances` that are contained in the instance.

Note that the path of an instance can be queried by concatenating all names of its instances in the `subInstance` containment hierarchy.

3.2.4 Class *ModuleInstance*

Instances defined by a module in the instance hierarchy.

Attributes

1. Attribute `name` (inherited from `NamedElement`) stores the instance name
2. Attribute `baseModuleName` holds the base module name from which this instance is instantiated. The base module can help to build libraries and enables reuse.
3. Attribute `abstractionType` indicates at which level the instance is (NL or RTL design):
    a. `NL`: shall be given for modules that only have instances and interconnects
    b. `RTL`: a module contains signal assignment, always-block, wait, delay, etc. The behavioral RTL and synthesizable RTL are not differentiated.

   c. `BEHAVIOR`: For all high-level blocks which are neither RTL or NL (e.g. used for SystemC descriptions)

4. Attribute `area` shall correspond to the *die area* of the instance, excluding sub-instances. It shall be a numerical value in the IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008) excluding any units. Note that all abstraction types can carry area information but depending on the design stage these might be (rough) estimates. The area information for each cell can be for example derived from Liberty files. Note the following:
   a. The unit of the area is not defined formally in the context of this specification. However, all instances of the same hierarchy shall use the same unit! An indication of the unit might be given or derived from the `technology` field, see section 3.2.2 for more information.
   b. Area of all cells in an instance shall be aggregated only if they are not listed as sub-instances. Instance area should not include area information of non-cell sub-instances.
   c. If an instance contains both cell instances and RTL blocks area information should not be included.
5. Attribute `cellCount` is the number of resulting cells in the chip layout of this instance. This number might be used if no further information is available (e.g. Liberty file not available).
6. Attribute `gateCount` is the number of gates this instance is containing
7. Attribute `sequentialElementsCount` is the number of sequential elements in the current instance (not including count of the sequential elements from its sub-instances). Note the following:
   a. For an RTL design, the number of sequential elements shall be counted in bit-level accuracy.
   b. For an RTL design with cell sub-instances, this number shall include the number of sequential UDPs from cell sub-instances.
   c. For a NL design, the number of sequential UDPs shall be included. Results shall also be in bit-level, FF arrays shall be expanded to get bit-level results.
8. Zero to multiple `moduleParameters` elements
9. Zero to multiple `ports` elements
10. Zero to multiple `subInstances` elements

3.2.5 Class *`ModuleParameters`*

Class `ModuleParameters` contains the instance parameters as list of name-value pairs.

Attributes

1. Attribute `name` (inherited from `NamedElement`) is the name of the parameter
2. Attribute `value` is the actual value of the parameter as String. The encoding of parameter values shall be according to W3C escaping so that basic XML syntax rules are not violated.

3.2.6 Class *`Port`*

Class `Ports` defines the port list of the instance.

Attributes

1. Attribute `name` (inherited from `NamedElement`) holds the name of the port
2. Attribute `direction` is the port direction. Values are `IN`, `OUT` or `IN_OUT` (compare enumeration `PortDirection`)
3. Attribute `numberOfBits` specifies the width of a port (digital bits)
4. Attribute `range` shows the actual range if the port is not scalar

3.2.7 Classes *Signal*, *SequentialCells*, *SequentialSignal*

Classes `Signal`, `SequentialCells`, and `SequentialSignal` are reserved for future extensions of signal-level fault injection campaigns.

3.3 FAULT INJECTION DATA

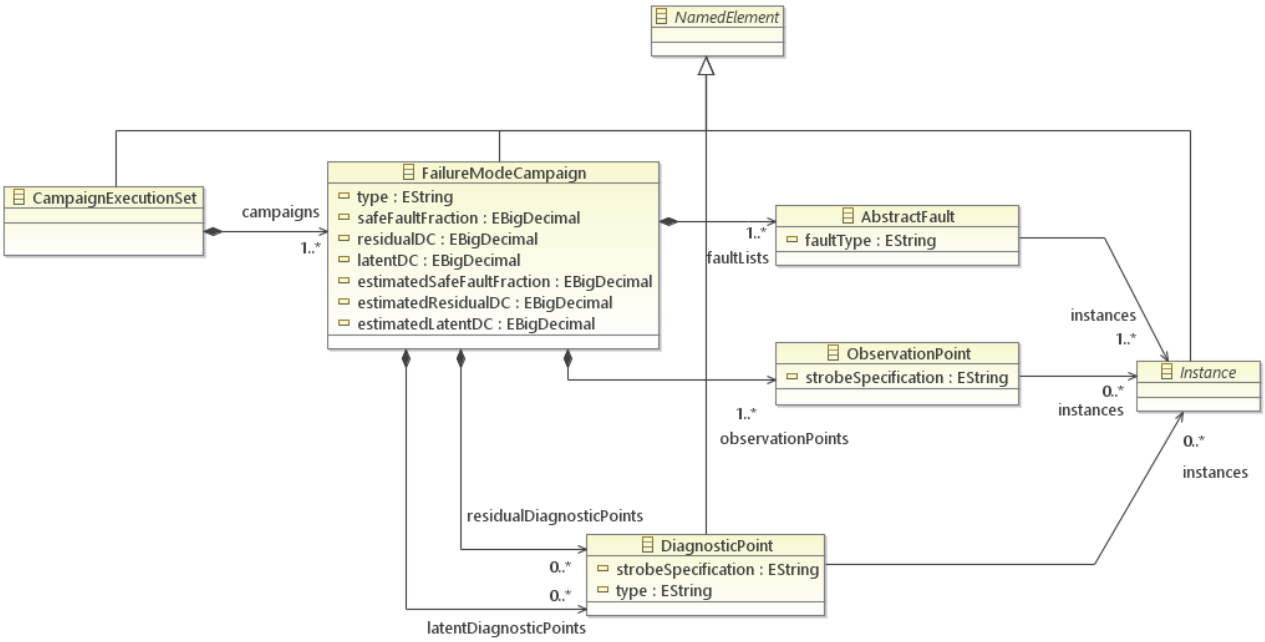

Figure 3. Metamodel for fault injection data

3.3.1 Class *CampaignExecutionSet*

For exchanging fault injection data, the class `CampaignExecutionSet` provides the top-level element in the exchange dataset. It is supposed to be used in conjunction with a hierarchy as defined in section 3.2.2.

3.3.2 Class *FailureModeCampaign*

Class `FailureModeCampaign` contains the fault injection campaign details.

Attributes

3. Attribute `name` (inherited from `NamedElement`) is the name of a failure mode under test (FMUT) for fault injection simulation
4. Attribute `type` is one of `PERMANENT` or `TRANSIENT` and describes the kind of failure mode under consideration. Note that this attribute is under consideration for future extension of more fault types (e.g. stuck-at-0, stuck-at-1, SEL, SET).
5. Attribute `safeFaultFraction` is percentage of safe faults determined by a fault injection campaign in the range [0..100] with an arbitrary precision
6. Attribute `estimatedSafeFaultFraction` is percentage of safe faults estimated in an FMEDA and passed to the fault injection tooling for plausibility checks (in the range [0..100] with an arbitrary precision)
7. Attribute `residualDC` is the percentage of DC determined for single point faults in the range of [0..100] with an arbitrary precision
8. Attribute `estimatedResidualDC` is an estimated percentage of DC for single point faults from an FMEDA that can be passed to the fault injection tooling for plausibility checks (in the range [0..100] with an arbitrary precision)

9. Attribute `latentDC` is the percentage of DC determined for single point faults in the range of [0..100] with an arbitrary precision
10. Attribute `estimatedLatentDC` is an estimated percentage of DC for latent faults from an FMEDA that can be passed to the fault injection tooling for plausibility checks (in the range [0..100] with an arbitrary precision)
11. Reference `faultLists` describes the faults to be injected with references to the instances where to inject them (meta-class *AbstractFault*). Typically, every failure mode is represented by exactly one abstract fault that references the extent for fault injection, i.e. all instances that build the cone of input for the given failure mode.
12. Reference `observationPoints` contains the definition where the FMUT is observed (as elements of class `ObservationPoint`). Attributes are described in section 3.3.4.
13. Reference `residualDiagnosticPoints` contains details where a diagnostic safety mechanism flags a detected fault (as elements of class `DiagnosticPoint`). Attributes are described in section 3.3.5.
14. Reference `latentDiagnosticPoint` contains details where a dual point fault diagnostic flags a detected fault (as elements of class `DiagnosticPoint`). Attributes are described in section 3.3.5.

3.3.3 Class *AbstractFault*

Class `AbstractFaults` defines where and how faults should injected during a fault injection campaign. This class defines the semantics of a failure mode by referencing the instances that shall be considered as cone of input for the failure mode (fault injection extent). If the fault injection patterns are complex or the failure mode consist of multiple faults, a list of `AbstractFaults` can be provided to reference different parts of the design.

Attributes

1. Attribute `faultType` defines technology dependent or tool specific extensions. The content is proprietary and allows vendors e.g. to refine different kinds of faults depending on technology, identify library blocks to be inserted as fault (e.g. analog blocks), or passing parameters to the engine for the given fault.
2. Attribute `instances` is a list of references into the hierarchy under `Scope` (see 2.2.1). All listed instances shall be included in the fault injection runs to determine the DC and safe fault fractions for the FMUT

3.3.4 Class *ObservationPoint*

Class `ObservationPoint` defines how the FMUT is observed as single point fault or latent faults.

Attributes

1. Attribute `strobeSpecification` (optional) to describe the output pattern to be observed that describes for FMUT for single or dual/latent point faults. If the output is digital, this value can be empty since a failure can be observed when it is different from the good run that has been executed for a FISC. For all other cases see section 3.4.
2. Reference `instances` is a list of references into the hierarchy under “Scope” (see 3.2.2). All listed instances together describe the observation point for the FMUT.

3.3.5 Class *DiagnosticPoint*

Class `DiagnosticPoint` defines how an FMUT can be detected by means of a safety mechanism.

Attributes

1. Attribute `name` (inherited from `NamedElement`) is the name of the safety mechanism implementing the diagnostic coverage

2. Attribute `type` defines the kind of safety mechanism that implements the diagnostic and is one of “HW” (Hardware) or “SW” (Software). Optional attribute, default is “HW” if omitted.
3. Attribute `strobeSpecification` (optional) to describe the output pattern to be observed that describes a detection of a fault. If the output is digital, this value can be empty. For all other cases see section 3.4.
4. Attribute `instances` is a list of references into the hierarchy under “Scope” (see 3.2.2). All listed instances together describe the diagnostic point for the single point fault or dual/latent point failure safety mechanism.

3.4 Strobe Specifications and Semantics

In order to make a verdict whether a failure mode occurred during fault injection or not, a comparison is required between a “good run” and subsequent runs with faults being injected. The required observation points are specified e.g. in an FMEDA and exchanged together with the FMUT as a set of references to ports, signals, or instances (see section 3.3). Based on the number and the kind of observation points, the following *failure strobes* are distinguished:

- *Single digital output:* If a single port with a bit width of 1 is given as observation point, any difference between expected output and measured output is considered a failure.
- *Semantics for digital bit width > 1.* If a single multi-bit port or multiple digital ports are referenced as observation point, by default any single bit difference is considered to represent a fault. If only certain bits make up a failure, an additional strobe vector specification can be included to distinguish the relevant bits:
    - The vector syntax is: “[“ *bit* | *range* “]”, with the expressions:
        - *bit* specifies the number of a single bit in the range of the complete multi-valued port bit vector
        - *range* is an expression of form “*n*:*m*” with *n* being the highest bit and *m* being the lowest bit, selecting the range of bits of interest

      Example:
        - A single bit selection of a port with 8 bit width is represented as "[7]", meaning only the 7th bit of the vector is indicating a difference
        - A range selection (multiple bits) is represented as “[3:0]” which means selecting only the lower 4 bits of the vector
- *Semantics for analog.* At the current stage, only digital outputs are covered.

Proprietary extensions to this definition might be applied by tool vendors.

4 XMI SCHEMA

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xsd:schema
        xmlns:ipd=http://www.ikv.de/medini/ipdesign/1.2
        xmlns:xmi=http://www.omg.org/XMI
        xmlns:xsd=http://www.w3.org/2001/XMLSchema
        targetNamespace=http://www.ikv.de/medini/ipdesign/1.2 >
  <xsd:import
        namespace=http://www.omg.org/XMI
        schemaLocation="platform:/plugin/org.eclipse.emf.ecore/model/XMI.xsd" />
  <xsd:simpleType name="PortDirection">
    <xsd:restriction base="xsd:NCName">
      <xsd:enumeration value="OUT"/>
      <xsd:enumeration value="IN_OUT"/>
      <xsd:enumeration value="IN"/>
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="Scope">
    <xsd:complexContent>
      <xsd:extension base="ipd:NamedElement">
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
          <xsd:element name="instances" type="ipd:Instance"/>
        </xsd:choice>
        <xsd:attribute name="topInstancePath" type="xsd:string"/>
        <xsd:attribute name="designVersion" type="xsd:string"/>
        <xsd:attribute name="technology" type="xsd:string"/>
        <xsd:attribute name="gateToAreaConversionFactor" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="Scope" type="ipd:Scope"/>
  <xsd:complexType abstract="true" name="NamedElement">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="name" type="xsd:string"/>
  </xsd:complexType>
  <xsd:element name="NamedElement" type="ipd:NamedElement"/>
  <xsd:complexType name="ModuleInstance">
    <xsd:complexContent>
      <xsd:extension base="ipd:Instance">
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
          <xsd:element name="moduleParameters" type="ipd:Parameter"/>
          <xsd:element name="subInstances" type="ipd:Instance"/>
          <xsd:element name="ports" type="ipd:Port"/>
        </xsd:choice>
        <xsd:attribute name="abstractionType" type="xsd:string"/>
        <xsd:attribute name="area" type="xsd:string"/>
        <xsd:attribute name="sequentialElementsCount" type="xsd:int"/>
        <xsd:attribute name="safeFaultsUT" type="xsd:string"/>
        <xsd:attribute name="baseModuleName" type="xsd:string"/>
        <xsd:attribute name="cellCount" type="xsd:int"/>
        <xsd:attribute name="gateCount" type="xsd:int"/>
      </xsd:extension>
    </xsd:complexContent>
```

```
</xsd:complexType>
<xsd:element name="ModuleInstance" type="ipd:ModuleInstance"/>
<xsd:complexType abstract="true" name="Instance">
  <xsd:complexContent>
    <xsd:extension base="ipd:NamedElement"/>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="Instance" type="ipd:Instance"/>
<xsd:complexType name="Port">
  <xsd:complexContent>
    <xsd:extension base="ipd:Instance">
      <xsd:attribute name="direction" type="ipd:PortDirection" use="required"/>
      <xsd:attribute name="numberOfBits" type="xsd:int" use="required"/>
      <xsd:attribute name="range" type="xsd:string"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="Port" type="ipd:Port"/>
<xsd:complexType name="SequentialCell">
  <xsd:complexContent>
    <xsd:extension base="ipd:ModuleInstance"/>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="SequentialCell" type="ipd:SequentialCell"/>
<xsd:complexType name="Parameter">
  <xsd:choice maxOccurs="unbounded" minOccurs="0">
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="value" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="Parameter" type="ipd:Parameter"/>
<xsd:complexType name="LibraryElement">
  <xsd:complexContent>
    <xsd:extension base="ipd:ModuleInstance"/>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="LibraryElement" type="ipd:LibraryElement"/>
<xsd:complexType name="AbstractFault">
  <xsd:choice maxOccurs="unbounded" minOccurs="0">
    <xsd:element name="instances" type="ipd:Instance"/>
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="faultType" type="xsd:string"/>
  <xsd:attribute name="instances" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="AbstractFault" type="ipd:AbstractFault"/>
<xsd:complexType name="ObservationPoint">
  <xsd:choice maxOccurs="unbounded" minOccurs="0">
    <xsd:element name="instances" type="ipd:Instance"/>
    <xsd:element ref="xmi:Extension"/>
  </xsd:choice>
  <xsd:attribute ref="xmi:id"/>
  <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
  <xsd:attribute name="strobeSpecification" type="xsd:string"/>
```

```
    <xsd:attribute name="instances" type="xsd:string"/>
  </xsd:complexType>
  <xsd:element name="ObservationPoint" type="ipd:ObservationPoint"/>
  <xsd:complexType name="FailureModeCampaign">
    <xsd:complexContent>
      <xsd:extension base="ipd:NamedElement">
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
          <xsd:element name="faultLists" type="ipd:AbstractFault"/>
          <xsd:element name="residualDiagnosticPoints" type="ipd:DiagnosticPoint"/>
          <xsd:element name="latentDiagnosticPoints" type="ipd:DiagnosticPoint"/>
          <xsd:element name="observationPoints" type="ipd:ObservationPoint"/>
        </xsd:choice>
        <xsd:attribute name="type" type="xsd:string"/>
        <xsd:attribute name="safeFaultFraction" type="xsd:string"/>
        <xsd:attribute name="residualDC" type="xsd:string"/>
        <xsd:attribute name="latentDC" type="xsd:string"/>
        <xsd:attribute name="estimatedSafeFaultFraction" type="xsd:string"/>
        <xsd:attribute name="estimatedResidualDC" type="xsd:string"/>
        <xsd:attribute name="estimatedLatentDC" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="FailureModeCampaign" type="ipd:FailureModeCampaign"/>
  <xsd:complexType name="CampaignExecutionSet">
    <xsd:complexContent>
      <xsd:extension base="ipd:NamedElement">
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
          <xsd:element name="campaigns" type="ipd:FailureModeCampaign"/>
        </xsd:choice>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="CampaignExecutionSet" type="ipd:CampaignExecutionSet"/>
  <xsd:complexType name="Signal">
    <xsd:complexContent>
      <xsd:extension base="ipd:Instance">
        <xsd:attribute name="numberOfBits" type="xsd:string"/>
        <xsd:attribute name="range" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="Signal" type="ipd:Signal"/>
  <xsd:complexType name="SequentialSignal">
    <xsd:complexContent>
      <xsd:extension base="ipd:Signal"/>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="SequentialSignal" type="ipd:SequentialSignal"/>
  <xsd:complexType name="DiagnosticPoint">
    <xsd:complexContent>
      <xsd:extension base="ipd:NamedElement">
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
          <xsd:element name="instances" type="ipd:Instance"/>
        </xsd:choice>
        <xsd:attribute name="strobeSpecification" type="xsd:string"/>
        <xsd:attribute name="type" type="xsd:string"/>
        <xsd:attribute name="instances" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
```

```
  </xsd:complexType>
  <xsd:element name="DiagnosticPoint" type="ipd:DiagnosticPoint"/>
</xsd:schema>
```

5 EXAMPLES

This section contains an example of a serialized instance hierarchy.

```
<?xml version="1.0" encoding="UTF-8"?>
<ipd:Scope xmlns:ipd="http://www.ikv.de/medini/ipdesign/1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xmi="http://www.omg.org/XMI" xmi:version="2.0" xmi:id="_1lzcgI7fEearZpgPKfZY4w" topInstancePath="top.DUT" designVersion="1.0">
   <instances xsi:type="ipd:ModuleInstance" name="dut" abstractionType="NL" area="15">
      <subInstances xsi:type="ipd:ModuleInstance" name="dut" baseModuleName="dut" abstractionType="NL" area="10" sequentialElementsCount="14">
         <ports name="clk" numberOfBits="1" direction="IN" />
         <ports name="cntr" numberOfBits="4" range="[3:0]" direction="OUT" />
         <ports name="reset" numberOfBits="1" direction="IN_OUT" />
         <subInstances xsi:type="ipd:ModuleInstance" name="M1" baseModuleName="mid" abstractionType="NL" area="9" sequentialElementsCount="7">
            <moduleParameters name="WIDTH" value="4" />
            <ports name="clk" numberOfBits="1" direction="IN" />
            <ports name="data1" numberOfBits="4" range="[3:0]" direction="IN" />
            <ports name="reset" numberOfBits="1" direction="IN_OUT" />
            <subInstances xsi:type="ipd:ModuleInstance" name="M1-Child" baseModuleName="low" abstractionType="NL" sequentialElementsCount="3">
               <ports name="clk" numberOfBits="1" direction="IN" />
               <ports name="ld" numberOfBits="1" direction="IN" />
               <ports name="reset" numberOfBits="1" direction="IN_OUT" />
            </subInstances>
         </subInstances>
         <subInstances xsi:type="ipd:ModuleInstance" name="M2" baseModuleName="mid" abstractionType="NL" sequentialElementsCount="5">
            <moduleParameters name="WIDTH" value="4" />
            <ports name="clk" numberOfBits="1" direction="IN" />
            <ports name="data1" numberOfBits="4" range="[3:0]" direction="IN" />
            <ports name="reset" numberOfBits="1" direction="IN_OUT" />
                        <subInstances xsi:type="ipd:ModuleInstance" name="M2-Child" baseModuleName="low" abstractionType="NL" area="6" sequentialElementsCount="2">
               <ports name="clk" numberOfBits="1" direction="IN" />
               <ports name="ld" numberOfBits="1" direction="IN" />
               <ports name="reset" numberOfBits="1" direction="IN_OUT" />
            </subInstances>
         </subInstances>
                  <subInstances xsi:type="ipd:ModuleInstance" name="M3" baseModuleName="mid" abstractionType="RTL" area="8" sequentialElementsCount="4">
            <moduleParameters name="WIDTH" value="8" />
            <ports name="clk" numberOfBits="1" direction="IN" />
            <ports name="data1" numberOfBits="4" range="[3:0]" direction="IN" />
            <ports name="reset" numberOfBits="1" direction="IN_OUT" />
                  </subInstances>
          </subInstances>
   </instances>
</ipd:Scope>
```

6 REFERENCES

[1] International Organization for Standardization: *ISO FDIS 26262:2018, Road vehicles – Functional safety*, Parts 1 to 11

[2] International Electrotechnical Commission: IEC 61508:2010, *Functional safety of electrical/ electronic/programmable electronic safety-related systems*, parts 1 to 5

[3] Object Management Group: *Meta Object Facility (MOF) Core Specification*, Version 2.5.1, formal/2016-11-01

[4] Object Management Group: *XML Metadata Interchange (XMI) Specification*, Version 2.5.1, formal/2015-06-07