Project Number: FP7-611404

D5.1.2 - Input parameters and system modeling formal representation

Authors[1]

A. Savino (POLITO), S. Di Carlo (POLITO), G. Di Natale (CNRS), A. Bosio (CNRS), T. Loekstad (ABB), M. Kalirorakis (UoA), S. Tselonis (UoA), N. Foutris (UoA), D. Gizopoulos (UoA), G. Politano (POLITO), M. Pipponzi (YOGITECH)

Version 1.3 – 31/10/2014

Lead contractor: Politecnico di Torino
Contact person: Alessandro Savino Control and Computer Engineering Dep. Politecnico di Torino, C.so Duca degli Abruzzi, 24 I-10129 Torino TO Italy E-mail: alessandro.savino@polito.it
Involved partners[2]**:** POLITO, UoA, CNRS, ABB, YOGITECH
Work package: WP5
Affected tasks: T5.1

Nature of deliverable[3]	R	P	D	O
Dissemination level[4]	PU	PP	RE	CO

[1] *Authors listed here only identify persons that contributed to the writing of the document.*

[2] *List of partners that contributed to the activities described in this deliverable.*

[3] ***R**: Report, **P**: Prototype, **D**: Demonstrator, **O**: Other*

COPYRIGHT

[4] ***PU**: public, **PP:** Restricted to other programme participants (including the commission services), **RE** Restricted to a group specified by the consortium (including the Commission services), **CO** Confidential, only for members of the consortium (Including the Commission services)*

INDEX

Scope of the document

This document is an outcome of task T5.1, "**Input parameters representation and standardization**", elaborated in the Description of Work (DoW) of the CLERECO project under Work Package 5 (WP5).

Figure 1 depicts graphically the goal of this deliverable, its main results, the inputs it uses and which work packages will use its outputs.

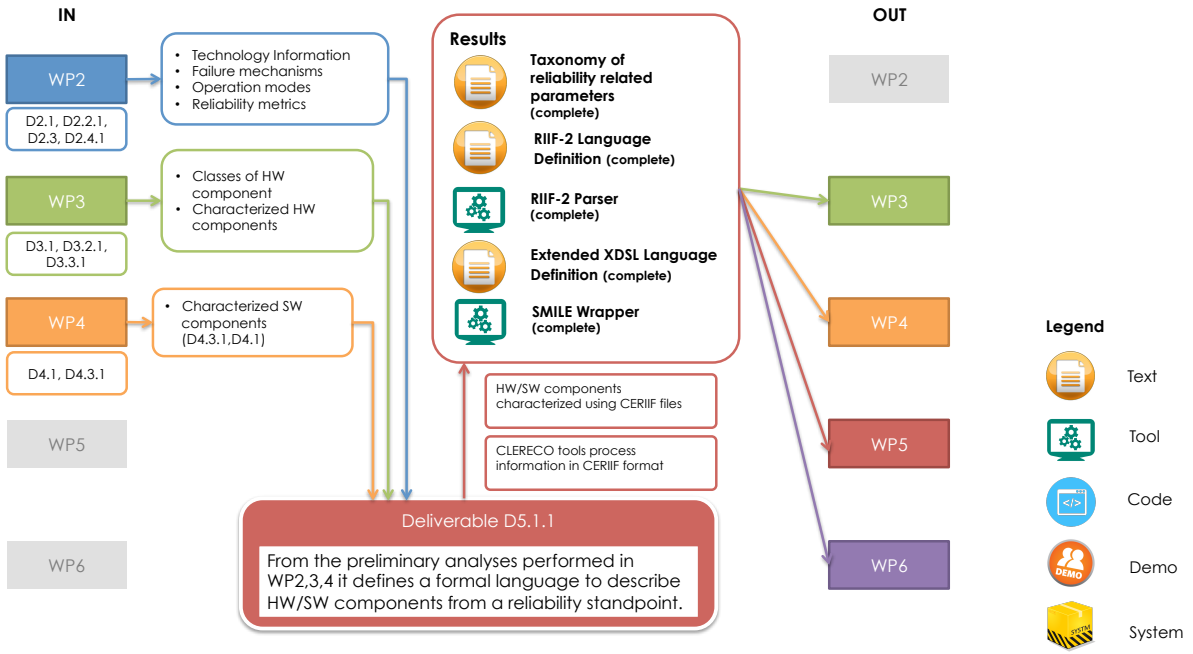


Figure 1 - Inputs and Outputs of this Deliverable

D5.1.2 has three main goals and outcomes:

1. The first goal is to provide the initial taxonomy of parameters associated with the components of a system that may potentially impact the reliability of the system. With the term **component** we consider both the hardware and the software components of the system, as described in Deliverable D3.1 (*Report on major classes of hardware components)* and Deliverable D4.1 (*Software Impact on system reliability: metrics and models*). Parameters considered in this document also include the failure mechanisms that may affect the selected components.
2. The second goal is to introduce a formal language for the representation of these parameters. This is required to enable their use within an Electronic Design Automation (EDA) tool. This represents an important step toward the implementation of a software framework for early reliability evaluation of complex systems.
3. The third goal is to propose a representation for the system architecture. This is mandatory to be able to analyze the system reliability within an EDA tool.

This document is a final update of the first version (D5.1.2). The main contribution if this final version is a consolidated description of the RIIF-2 language for the description of single

components and the introduction of the extended XDSL language used to model the architecture of the system.

The document is organized in the following sections:

- **Introduction.** This section shortly overviews background research on reliability parameters and standardization of reliability related information.
- **System components characterization.** This section introduces the way reliability information of each components are described resorting to the RIIF-2 language defined in the framework of the project
- **System description language.** This section describes how the system architecture is represented resorting to the extended XDSL language defined in the framework of the project.
- **Conclusions.** In this final section, we summarize the work done for the deliverable and we set a roadmap to reach a full reliability-oriented system description.

1. Introduction

Nowadays, reliability information for complex digital systems is mainly confined to the technology and circuit level. At these levels, there is a deep knowledge of the different failure mechanisms. However, cross-layer reliability requires a comprehensive full system model that includes all failure types and the propagation of their effect to higher levels of the system stack. This is a fundamental requirement to enable improved methodologies for analyzing the reliability of systems built from unreliable components and process technologies. Among them the most challenging problem is to properly abstract information about reliability issues that was gained at the technology level. One key aspect is to define the correct interface to propagate reliability information up in the design abstraction levels. This interface must be designed in such a way that important knowledge on reliability can be linked through levels.

An overly simplified reliability description formalism often used for system level reliability analysis is Reliability Block Diagrams (RBDs) [1][2]. Although RBDs cannot be classified as a language for reliability information management, they allow for simple descriptions and characterization of a system. Each block in a RBD represents a system component with an associated failure rate. The structure of the RBD defines the logical interactions of failures within a system that are required to sustain correct system operation.

Focusing on the way reliability related information could be described, we observed an increased interest, within the research community, in the definition and standardization of reliability oriented description languages [7]. This is an important task for the CLERECO project, where CLERECO is going to deliver important contributions. Representing reliability related information in a proper way is essential to distribute the reliability analysis throughout the design flow of a system and to propagate information across different levels of the system's hierarchy.

Only a few publications focus on system's modeling for reliability analysis at the hardware layer [1][4][5]. A system is mainly modeled for simulation purposes in which the occurrence of a fault is emulated and its propagation within the system is analyzed. Relevant parameters that must be modeled in this scenario are limited to the fault properties (i.e., time, feasible locations, etc.), and the final reliability metrics computed based on the simulation results.

Some of the first attempts to model reliability information are reported in [8][9][10]. Even if these papers are still not working in a cross-layer scenario, the main idea is to split the system into a set of interrelated blocks that share information about the reliability of individual components. While representing a first improvement, the main drawback of these approaches is that they oversimplify the description of a system limiting reliability related information to simple fault rates.

A main step toward modeling reliability information of a system's component has been recently proposed in [6] through the Reliability Information Interchange Format (RIIF). Although limited to hardware components, RIIF has a set of primary characteristics that fit what is needed in CLERECO for the reliability-related description of a system's component.

In this deliverable we aim at identifying a reliability description languages able to:

- Describe how specific failure modes are affected by specific functional parameters of the component (e.g., voltage, size, etc.).
- Enumerate the failure modes of a component.
- Build composite components from simpler components.
- Be scalable from cell level through to system-level.
- Be general-purpose (not tied to a single application or system architecture).
- Provide a mean to standardize the modeling of generic components (e.g., DRAMs) using templates.
- Specify reliability targets that must be met.

Limiting the description to the hardware domain contradicts the main objective of the CLERECO project. Since the full system is composed of both hardware and software components, the defined description languages must be general enough to work with both hardware and software components and to link information among layers in order to properly describe how errors propagate within the system.

This deliverable will cover two main aspect in cross-layer reliability information description. First it will focus on the definition of a proper formalism to describe reliability information for hardware and software components of the system. Second, it will focus on an appropriate formalism to represent the system architecture that models how the different components are connected together.

2. System components characterization

2.1. Taxonomy of Components Reliability Parameters

The CLERECO project has two dedicated work packages aiming at characterizing reliability aspects of hardware (WP3) and software (WP4) components of a system. This section starts from the results of these two work packages to create a taxonomy of relevant parameters that characterize a component in terms of its impact on the overall system's reliability. The main goal of this taxonomy is to identify similarities and differences among classes of components in order to be able to provide a compact and formal description of each component at the system level.

At a first glance, hardware and software components will be studied separately in order to analyze and highlight their peculiar characteristics. Common parameters of these two macro classes will be later merged in order to simply the system's description formalism. The provided taxonomy is not meant to be exhaustive. It serves as a starting point for the definition of a dedicated description language, which will be flexible to enable easy updates and add-ons.

In order to have a uniform description of the identified parameters, each reliability-related parameter will be described in terms of the following information items:

- **Label**: a keyword identifying the parameter.
- **Description**: a free text describing the meaning and use of the parameter.
- **Data Type**: the parameter's data type (e.g., integer, string, etc.) required to identify how the related information can be stored.
- **Domain**: the set of accepted values for the parameter.
- **Unit**: the measurement unit for the parameter (if applicable).
- **Mandatory**: a flag indicating whether the parameter is *optional* or *mandatory*.

2.1.1. Hardware Components Description

Table 1 summarizes the list of parameters identified for the characterization of a hardware component of a system. The list includes either generic parameters required to identify the component as well as more specific parameters modeling reliability related aspects of the component.

According to deliverable D3.1 (*Report on major classes of hardware components)* hardware components are classified in CLERECO into five main categories: (1) Microprocessors, (2) Accelerators, (3) Memories, (4) Peripherals and (5) Interconnections. The *Class* parameter is used to place a component within one of these five major classes. Moreover, the *Subclass* parameter enables to further refine the component's classification defining subclasses within the five macro-classes (e.g., within the Memory macro class, a component can be further classified into a specific memory type including flash memories, SRAM, DRAM, etc.). Technological information analyzed in WP2 such as the technology process, the node size and the component area are among the most important parameters to define the reliability level of a hardware component and are therefore included in the list of considered parameters together with higher level architectural parameters (e.g., protection mechanisms) and functional parameters (e.g., set of instructions or operations implemented by the component).

Table 1 - Hardware Component Parameters

Label	Description	Data Type	Domain	Unit	Mandatory
Name	Component's name	String	-		YES
Vendor	Component's Vendor Name	String	-		YES
Type	Set to HW to identify hardware components	String	{HW, SW}		YES
Class	Component's class according to Deliverable D3.1.	String	{Microprocessor, Accelerator, Memory, Peripheral, Interconnection}		YES
Subclass	Component's subclass, if needed to distinguish among components of the same class	String	-		NO
Technology	Information about the technology process used to implement the component according to Deliverable D2.1.	String	{CMOS, FinFET, …}		YES
Node Size	Technology node size dimension	Number		{µm, nm, …}	YES
Area	Component area.	Number	-	{mm^2, gates, bits, …}	YES
Word Length	The number of bits considered as a word for the operations (if defined). According to D3.2.1, the wider the word length the higher is the vulnerability of the component.	Number	-	-	NO
ATPG-difficulty	A value to identify how difficult is generating ATPG test patterns. According to D3.2.1, hard to detect faults can slip into production more easily.	Number	-	-	NO

Inherent Redundancy	The amount of occurrences of subcomponents to identify per-se fault tolerant architectures, according to D3.2.1.	List	-	-	NO
Operation Set	Set of all available operations.	Table	See Table 2		NO
Error Rates	List of error rates information about the component	Table	See Table 3		YES
Protection Mechanisms	List of error protection mechanisms implemented by the component	Table	See Table 4		NO

Most hardware components employed in modern digital systems are able to perform well-defined and structured operations (e.g., instructions implemented by microprocessors and accelerators, read/write operations implemented by memory blocks, data transactions implemented by interconnection infrastructures, etc.). Different operations may generate different behaviors in case of faults, thus leading to fault masking effects or fault amplification effects. To properly describe the operations implemented by a component the Operation Set parameter describes a list of available operations each one represented according to the information items reported in Table 2.

Table 2 - Operation Set Attributes

Label	Description	Data Type	Domain	Unit	Mandatory
Name	The operation Name	String	-	-	YES
Type	The operation type. Helps clustering operations (e.g., mathematical operations)	Sting	-	-	YES
Timing /Latency	The expected timing.	Number	-	{clock cycles, seconds, … }	YES
Involved Area	If available the portion of area implementing the operation	Number	-	{mm^2, gates, bits, flip-flops,…}	NO
Fault Models Masking Probabilities	If a set of fault models has been investigated, masking probabilities related to them may be available. They could generate one or more attributes (one for each probability)	Number	-	-	NO

When dealing with reliability related information, components are usually characterized in order to understand their sensitivity to a selected list of Fault Models (FMs), providing *Error Rates* for each of the considered FM. This list of error rates represents one of the most important information for systems developers to understand the impact of a component on the reliability of a system. Detailed and accurate error rate information for each component represents the starting point to identify efficient reliability evaluation strategies. The error rate is usually linked to a FM. Its value can be either provided as an absolute value or through the definition of a mathematical model that enables to compute the error rate based on a set of related variables, e.g., the area of the component, particles strike statistical rate, etc.

Along with failures investigation, components may also be designed to include dedicated *Protection Mechanisms* able to increase the component's reliability. A protection mechanism (detection, diagnosis, recovery, repair) is in general able to mitigate the effect of selected types of fault models. Moreover, a protection mechanism could be specifically designed to protect only a set of operations of the component, e.g., the ones heavily affected by faults. The effect of the protection mechanism can in general be mathematically modeled as a modification of one of the raw error rates defined for the component.

Table 3 and Table 4 show the main attributes identified to describe both the Error Rates and the Protection mechanisms.

Table 3 - Error Rates Attributes

Label	Description	Data Type	Domain	Unit	Mandatory
Fault Type	The Error Rate type.	String	{permanent, intermittent, transient}	-	YES
Fault Model	The Related Fault Model	String	{stuck at, single bit upset, …}	-	YES
Rate Model	The rate value. Usually a formula taking into account several variables to compute the value or a single value.	String /Number	-	{FIT, MTBF,…}	YES
Timing Model	Since each fault may introduce an effect not only in the output but also in the operation timing, a modal of that impact could be provided.	String /Number	-	{clock cycles, seconds, …}	NO

Table 4 - Mitigation Mechanism Attributes

Label	Description	Data Type	Domain	Unit	Mandatory
Type	The Mitigation Mechanism type.	Sting	-	-	YES
Affected Fault Models	The list of all affected fault models	List	-	-	YES
Affected Operations	The list of all affected operations	List	-	-	NO
Rate Model	The Rate model of the mechanisms. Usually a formula to compute the effect of the mechanism by evaluating several variables	String / Table	-	-	YES
Timing Model	Since the mitigation mechanism could introduce timing effects (i.e., a computation delay), it should be described here	String / Table	-	-	NO

In order to clarify the hardware description, Table 5 provides a set of available information for an instance of the OpenRISC 1200 microprocessor, publicly available on the Opencores.com website [27], organized as explained before. To keep the description short, in this document we only report a very small subset of instructions, two fault models and one protection mechanism.

Table 5 - OpenRISC 1200 Component characterization example

Label	Data	Unit
Name	OpenRISC 1200	-
Vendor	Opencores.org	-
Type	HW	-
Class	Microprocessor	-
Subclass	RISC	-
Technology	CMOS	-
Node Size	0.18	µm
Area	0.5	mm^2

Operation Set

Name	Type	Timing	Involved Area	SBU Fault Mask Probability	Stuck-At Mask Probability
ADD	add instruction	1	3 * Registers Flip-Flop Size	0.001	0.001
BNE	branch instruction	1	Registers Flip-Flop Size	0.015	0.015
MULTU	multiply instruction	10	4 * Registers Flip-Flop Size	0.25	0.10
SLL	shift instruction	1	2 * Registers Flip-Flop Size	0.001	0.001
…	…	…	…	…	…

Error Rates

Type	Fault Model	Rate	Timing Model
Permanent	Stuck_At	RAW Stuck_At Probability * (1 - Operation Stuck_At Masking probability)	NA
Transient	Single Bit Upset (SBU)	SBU probability * (Operation Involved Area / Microprocessor Area)	Operation Timing + 25%
…	…	…	…

Protection Mechanisms

Type	Affected Fault Models	Affected Operations	Model
Triple Module Redundancy (TMR)	Stuck At	All	Operation Time + (TMR computation & voting time)
…	…	…	…

2.1.2. Software Components Description

Software components are quite difficult to profile. They can be characterized by static properties that can be obtained by statically analyzing the software code without actually executing it, or by dynamic properties collected during the actual execution of the software. Dynamic properties are particularly difficult to collect since they are strongly influenced by the

software workload (i.e., the set of inputs provided to the software) used during the analysis of the component.

Table 6 summarizes the list of parameters identified for the characterization of a software component within a system, which is derived from the software characterization activities described in Deliverable D4.2.2 (*Software Characterization Methods*). The reader may notice that some of them overlap (e.g., name, vendor, type, class, subclass, etc.) with parameters defined for hardware components. This goes in the direction of trying to have a uniform and coherent description of all system's components.

Table 6 - Software Component Parameters

Label	Description	Data Type	Domain	Unit	Mandatory
Name	Component's name	String	-		YES
Vendor	Component's Vendor Name	String	-		YES
Type	Set to SW to identify hardware components	String	{HW, SW}		YES
Class	Component's class.	String	{Application, OS}		YES
Subclass	Component's subclass, if needed to distinguish among components of the same class	String	{Library, Device Driver, …}		NO
Size	This parameter characterizes the component size.	Number	-	{Line of code, instructions, executable size, etc.}	NO
Reading Access Rate	The count of memory read operations. It can be retrieved either from static or dynamic analyses. Its actual value must be linked to the access that could be generated to memory or cache.	Number	-		NO
Writing Access Rate	The count of write operations. As for the read accesses, this information can be evaluated by	Number	-		NO

	static or a dynamic analysis.				
Memory Accesses	The count of real memory accesses. It can be evaluated only by **dynamic** analysis using several workloads.	Number			YES
Cache Misses	The count of cache misses. If the information is available during a dynamic analysis, it counts the cache misses in case of memory accesses	Number			NO
Cache Hits	The count of cache hits. As for Cache misses, if available.	Number			NO
Touched Memory Pages	The count of memory pages touched by the component's memory accesses.	Number			YES
Loops number	The number of loops in the software	Number			NO
Variables Lifetime	The expression of the variable lifetime. It could be an average value or a distribution function	Number / String	-	-	NO
Algorithm Complexity	The complexity of the component, i.e., computing the number of nested loops…	String			NO
Timing Constraints	The list of all possible timing constrains.	List	See Table 7		NO
Software Faulty Behaviors	The correlation between Fault Models and the observed Software Faulty Behaviors	List	See Table 8		YES

The SW components often have timing constraints. A program or software routine is expected to end and to provide some outputs. Reliability issues may affect the software timing. Therefore, it is important to be able to define *Timing Constraints* when characterizing a software component. Since the execution time of a software component always depends on the

software workload, timing constraints are here defined in relation to a given workload as shown in Table 7. They rely on two basic data: the *Workload* and the *Expected Execution Time*. Moreover, if margins can be accepted in the execution time the optional *Max Accepted Execution Time* and *Average Accepted Execution Time* properties can be used.

Table 7 - Timing Constraints Attributes

Label	Description	Data Type	Domain	Unit	Mandatory
Workload	The Workload used as reference	String	-	-	YES
Expected Execution Time	The expected execution time. Usually, provided by the developer.	Number	-	{clock cycles}	YES
Maximum Accepted Execution Time	The maximum execution time that can let consider the component as correctly working (even if later than expected).	Number	-	{clock cycles}	NO
Average Accepted Execution Time	The average execution time, computed resorting to several runs.	Number	-	{clock cycles}	NO

Eventually, we may need information about the classes of Software Faulty Behaviors (SFB)[5] associated to the component. In this case, a list of Software Fault Models (SFM) and occurring SFBs must be provided. They will help, at a first glance, to define among all SFBs the ones of interest during reliability estimation. The reader may refer to deliverable D4.1 for a detailed taxonomy of SFMs and SFBs identified in the project.

Table 8 - Software Faulty Behaviors Attributes

Label	Description	Data Type	Domain	Unit	Mandatory
Fault Type	The Software Fault Model Type. They are similar to the HW ones.	String	{transient, intermittent, permanent}	-	YES
Fault Model	The Related Software Fault Model.	String	-	-	YES
Occurring SFB	The list of all expected Software Faulty Behaviors.	List	-	-	YES
Occurring	The occurrence prob-	List	-	-	YES

[5] *See section 4 of Deliverable D4.1 (*Software Impact on system reliability: metrics and models*) for more details.*

SFB Probabilities	ability for each SFB

In order to better understand the software component description, Table 9 provides an example of description of a simple software application performing the sum of two vectors. Fault injection has been conducted on this application to extract useful reliability information[6]. Rates are calculated with respect to a maximum run of 10000 elements in the vector.

Table 9 - Software Component Characterization example

<table>
<tr><th>Label</th><th>Data</th><th>Unit</th></tr>
<tr><td>Name</td><td>ADD Vector Application</td><td>-</td></tr>
<tr><td>Vendor</td><td>CLERECO</td><td>-</td></tr>
<tr><td>Type</td><td>SW</td><td>-</td></tr>
<tr><td>Class</td><td>Application</td><td>-</td></tr>
<tr><td>Subclass</td><td>Vector Operation Algorithm</td><td>-</td></tr>
<tr><td>Size</td><td>453</td><td>-</td></tr>
<tr><td>Reading Access Rate</td><td>76 * # of vector element / 10000</td><td>-</td></tr>
<tr><td>Writing Access Rate</td><td>75 * # of vector element / 10000</td><td>-</td></tr>
<tr><td>Memory Accesses</td><td>151 * # of vector element / 10000</td><td>-</td></tr>
<tr><td>Loops Number</td><td>3</td><td>-</td></tr>
<tr><td>Algorithm Complexity</td><td>N</td><td>-</td></tr>
<tr><td>Timing Constraints</td><td colspan="2"><table>
<tr><th>Workload</th><th>Expected Execution Time</th><th>Maximum Accepted Execution Time</th><th>Average Accepted Execution Time</th></tr>
<tr><td>Test Bench #1</td><td>10^{-6}</td><td>2</td><td>10^{-5}</td></tr>
<tr><td>…</td><td></td><td></td><td></td></tr>
</table></td></tr>
<tr><td>Software Faulty Behaviors</td><td colspan="2"><table>
<tr><th>Fault Type</th><th>Fault Model</th><th>Occurring SFB</th><th>Occurring SFB Probabilities</th></tr>
<tr><td>Permanent</td><td>Wrong Data</td><td>In-Time, Undetectable, Early, Late, Responsive, Full Unresponsive, Partially Unresponsive, Data Benign, No Data, EDC, Non-EDC</td><td>0.893, 0.107, 0, 0, 0.891, 0.42, 0.67, 0.413, 0.109, 0.052, 0.426</td></tr>
<tr><td>Permanent</td><td>Instruction Replacement</td><td>In-Time, Undetectable, Early, Late, Responsive, Full Unresponsive, Partially Unresponsive, Data Benign, No Data, EDC, Non-EDC</td><td>0.274, 0.726, 0, 0, 0.378, 0.348, 0.274, 0, 0.726, 0, 0.274</td></tr>
<tr><td></td><td>…</td><td></td><td></td></tr>
</table></td></tr>
</table>

[6] *For further details refers to Deliverable D4.2.2 (Software Characterization Methods)*

2.2. Components Description Language

Once a set of relevant parameters has been identified in order to characterize reliability of HW and SW components in a system, these parameters must be described exploiting a description language that enables easy access of this information in the reliability evaluation EDA tools developed within CLERECO.

A language for reliability information description and system interaction modeling needs to include certain features to enrich the description:

- *Definition of reliability oriented keywords*. Reliability has well defined parameters that must be associated to predefined keywords in the language. Therefore, reliability oriented languages, if available, will be preferred over general-purpose language to build as a starting point for the activities of the project.
- *Template mechanism*. HW and SW components can be in general clustered into classes that share similar information. Defining templates of components could be helpful. It can potentially reduce the time required to describe a new component and improve the correctness and consistency of the description.
- *Inheritance mechanism*. Components can be often classified into families that share overall characteristics with small differences (e.g., different models of a single microprocessor). An inheritance mechanism will reduce redundancy in the description by describing at each hierarchical level only the information that help differentiating the component from the higher level. A clear drawback is that, resorting to inheritance, the readability by humans will be more complex.
- *Values as formula*. While most reliability parameters assume exact values, some of them can be defined as function of other parameter. A reliability description language must be able to support this.
- *Reliability related data types*. General-purpose languages define very simple data types. To manage reliability information, reliability oriented data structures must be available in the language.
- *HW and SW description*. Since CLERECO takes into account the whole system's stack, it is mandatory to have a language general enough to describe characteristics of both hardware and software components.

This section reviews a set of already exiting languages for *information modeling* (both general purpose and reliability oriented) with the goal to identify a candidate language to serve as a starting point for the definition of a component description language in CLERECO. Our analysis also takes into account the possibility of re-using tools that have already been developed, thus reducing the effort to coding effort in the project. Specifically, we look for:

- *The availability of (open source) parsers*.
- *Extensive language documentation.*

The languages considered in this preliminary analysis are:

1. The Extensible Markup Language (XML).
2. The Unified Modeling Language (UML).
3. The Reliability Block Diagram (RBD).
4. The Reliability Information Interchange Format (RIIF).

2.2.1. XML

The Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It is defined in the XML 1.0 Specification produced by the World Wide Web Consortium (W3C) [15]. It is a textual data format with strong support via Unicode for different human languages. The design goals of XML emphasize simplicity, generality, and usability, specifically addressing In-

ternet as final platform. The design of XML focuses on documents but it is widely used for the representation of arbitrary data structures, [16].

XML, as a markup language, enables to describe any type of required keyword (so called *tags* in the XML syntax). However, it does not provide any *direct* template or inheritance mechanism. The only way to introduce templates and inheritance would be resorting to some intermediate representation of the information, such as the Document Object Model (DOM) [17]. This means building the mechanisms beyond the description, which seems a rather useful feature. Looking at the ability of describing values and their metrics, the language supports complex descriptions of values by resorting to tags and tag attributes. In XML, tags can be freely defined, it is therefore feasible to properly describe HW and SW components and let the user to define its own keywords. Serious concerns arise when formulas need to be defined instead of precise values. The ability of the language to describe formulas is out of discussion but the compliance with actual parsers must be verified.

In terms of tools and documentation, thanks to its wide diffusion, XML is quite well supported and large amount of information can be found on the Internet.

2.2.2. UML

The Unified Modeling Language (UML) offers a way to visualize the system's architectural design in a diagram, including elements such as activities (jobs), individual components of the system, and the interaction among components. Although originally intended solely for object-oriented design documentation, the UML has been extended to cover a larger set of application fields. Its general-purpose structure makes it suitable for the description of both HW and SW system's components. Nowadays the UML is adopted and managed by the Object Management Group (OMG) and it is an ISO standard, [18].

Regarding the reliability context, UML is a general-purpose language and modeling approach. Therefore, no reliability keywords are defined in the language, but they can be easily defined in the form of variables within a component. Moreover, UML allows both templates and inheritance because it follows the object-oriented paradigm [19]. A huge limitation stems in the possibility of defining metrics as well as using formulas instead of values for given parameters. Within classes, variables and processes are the only elements that can be described and no further extension is easy to plan.

While UML is more complex compared to XML, a very wide and active community guarantees the availability of parsers, tools and abundant documentation.

2.2.3. RBD

Reliability Block Diagrams (RBDs) do not belong exactly to the class of description languages but since they are largely use in the reliability evaluation [20][21], the CLERECO project takes them into account. An RBD is a diagrammatic method to analyze large and complex systems using block diagrams to show network relationships and to exploit how component reliability contributes to the success or failure of a complex system. RBD is also known as a dependence diagram (DD). Each block represents a component of the system with a failure rate. The structure of the RBD defines the logical interactions of failures within a system that are required to sustain system operation.

While the application context is the reliability of systems, RBD do not offer high flexibility in the characterization of single components. Usually, the block description is limited to the failure rate of each component. The RBD simplicity also means that it does not support template and inheritance mechanisms, along with metrics associated to the parameters and formulas instead of values. This very small set of information may require extending the RBD description, basically completely changing the language. Since the actual version of the language speci-

fies failure rates only, modeling of HW and SW components within the same system is not challenging.

There is a quite large set of commercial tools exploiting RBD descriptions while we observed a general lack of open source software and libraries. A set of commercial tools exploiting RBD for reliability analysis can be found in Section 4 of Deliverable D7.4.1 (Exploitation Plan Version 1).

2.2.4. RIIF

The *Reliability Information Interchange Format* (RIIF) is an application-specific language targeting the problem of modeling failure propagation in System on Chips (SoCs). RIIF was first proposed in [6] and was further developed during a dedicated workshop at Design and Test in Europe Conference 2013 (DATE'13) [24][25]. It expresses the failure mechanisms associated with a generic hardware component. Complex components can be built by combining simpler components and the propagation of failures from lower to higher levels can be expressed.

The language already includes reliability keywords helping the description of failure mechanisms and their propagations (e.g., failure rates can be express either as a single value or a formula). Moreover, each parameter can be defined including the *unit* (keyword for metric) associated with, and its value can be a formula expressing it as a function of other parameters. Since RIIF has been developed taking into account real use cases, it offers a very rudimental approach to template and inheritance mechanisms. However, this mechanism is quite simple and may require significant improvements. The language usage is focused on HW components, thus including SW components may require extending the language.

Very recently a Java tool including a command-line interface to read, parse, calculate, navigate and write RIIF files has been released [26]. Although it is a very limited version, it comes under an open source license, thus it can be extended and maintained open to the community. On the other hand, documentation is still very poor, mainly related to the few papers already published, [1][23][24][25]. The early stage of development of the RIIF language brings an opportunity to the CLERECO project, which most probably must be investigated.

2.2.5. Languages comparison

Table 10 proposes a general comparison of the characteristics of all languages overviewed in the previous sections.

Table 10 – Reliability Languages Investigation Comparison Summary

Characteristics	Language			
	XML	**UML**	**RBD**	**RIIF**
Reliability Keywords	NO	NO	YES	YES
Templates	NO	YES	NO	PARTIAL
Inheritance	NO	YES	NO	PARTIAL
Formulas	YES	NO	NO	YES
Metrics	YES	NO	NO	YES
HW & SW	YES	YES	YES	NO
Parser	YES	YES	NO	YES
Documentation	YES	YES	NO	NO

From the analysis of the table it is clear that RIIF is the language that fits more requirements than other languages. Going into details, more than the others, RIIF conjugates the flexibility of a general-purpose language with the ability of dealing with specific reliability related parameters (see [23] for more examples). The built-in ability of defining values as a function of other parameters, the possibility of specifying measurement units associated to a parameter's value and the strong focus on real use cases, suggest that the language can be improved to fit all CLERECO requirements with reasonable effort. Moreover, since the reliability community seems to support RIIF as the new generation language to model and describe systems and components in the reliability context [23], RIIF seems to be a very good candidate as base language for the CLERECO project. The lack of documentation is of course a main obstacle to the use of this language that may impact on the learning curve compared to other languages. Nevertheless, the current version of the language is in a very early development stage and further documentation can be expected with the next release possibly including improvements developed within the CLERECO project. To mitigate this risk a strict collaboration with iROC Technologies, which has first presented the RIIF language has been established in order to work on a joined extension to the language.

Since CLERECO is going to investigate reliability estimation models with a larger scope compared to the one considered in RIIF (e.g., RIIF focuses on HW components only) within task T5.1 there was an effort to improve the original language with a set of extensions to meet the specific CLERECO need. The resulting language has been named **RIIF-2**. This extended language will be described in the following sections

2.3. RIIF-2 definition

The goal of the RIIF-2 language is twofold: (i) enabling new, powerful language structures able to cope with the complexity of the full system stack, and (ii) extending the RIIF description capability to the software components of the system. The RIIF language is extended in order to provide additional flexibility in the description by introducing new keywords and statements and by broadening the usage of some already defined language mechanisms. In particular, the following extensions have been introduced:

- *An advanced template mechanism*. In order to exploit modularity and reuse for generic components, it is important to ensure that description of similar modules (e.g., SRAMs from different suppliers) is consistent. RIIF already includes a simple template mechanism to accomplish this goal, which is extended with dedicated statements to improve the readability of the language and to allow for complex uses such as implementation of multiple templates from a single component.
- *A full inheritance mechanism.* Components can be often classified into families that share overall characteristics with small differences (e.g., different models of a single microprocessor). The availability of an inheritance mechanism will significantly reduce redundancy in the description of families of components enabling for optimized information management, and reduced risk of modeling errors.

Complex data structures. Reliability information may require data aggregation to ease recurrent operations during computational activities such as failure rates evaluations. Complex data structures introduced in the RIIF-2 language include associative arrays and clustered data in the form of tables. Moreover a new indexing operator to easily access subsets of data in a table is proposed.

2.3.1. Brief RIIF Overview

This section reviews the basic RIIF concepts. Interested readers may refer to [6] for a detailed description of the initial language. In RIIF, the keyword **component** is used to model a system

component representing the main RIIF entity. Together with the component two additional types of *entities* are available:

1. the reliability requirements that a component needs to meet (**requirement** keyword),
2. the environment under which the component is going to operate (**environment** keyword).

Figure 2 reports the RIIF syntax to define components, requirements and environments where <LABEL> is a unique name to identify the instance of the entity.

```
component <LABEL>;
  …
endcomponent
```

```
requirement <LABEL>;
  …
endrequirement
```

```
environment <LABEL>;
  …
endenvironment
```

Figure 2 RIIF component, requirement and environment definitions

To parameterize *entities*, RIIF offers two alternatives (keywords): *constants* (**constant** keyword) and *parameters* (**parameter** keyword).

The main difference is that constants express static values whereas parameters express variable values computed as a function of other constants and parameters. Constants are in general used to describe constant internal information of the component while parameters are in general used to describe information exposed by the component to the other components.

In terms of reliability, within a component, the user is able to declare different failure modes (**fail_mode** keyword) and their rate of occurrence can be expressed as a function of any other already defined parameters or other failure modes.

The following RIIF snapshot proposes an example of basic usage of constants and parameters applied to the description of a register file component.

```
component REGISTER_FILE;
    …
    parameter NUMBER_REGISTERS: integer := 8;
    parameter FF_PER_REG: integer := NUMBER_REGISTERS * 32;
    constant SBU_TEMPERATURE_EFFECT_COEFF: float := 5.6e-12;
endcomponent
```

Both parameters and constants must define the **type** of the associated information. The syntax to define a parameter or a constant is based on the following code:

```
    <keyword> <label>: <type> [:= <value>];
```

Basic data types defined by RIIF are: **boolean**, **integer**, **float**, **enum** (as for an enumerative of items), and **time** (to define timing related information). The value of is optional. It can be set later in the definition and the actual value can be either explicit or a formula (FF_PER_REG in the example is expressed in terms of NUMBER_OF REGISTER value).

Each RIIF entity (component, environment, requirement) offer a getValue function that can be used to retrieve the falue of constant and parameters for the component as reported in the following code:

```
  parameter CURR_TEMPERATURE: float;
  assign CURR_TEMPERATURE'value = environment.getValue(TEMPERATURE);
```

In this case, the parameter CURR_TEMPERATURE is linked to the *environment* (defined elsewhere), which owns a parameter TEMPERATURE. The example also highlights how values to parameters can be associated after their definition suing the assign keyword:

```
    assign <label>'<attribute> = <value>;
```

The *assign* keyword set a value to an attribute of a parameter (referred through its label). Attributes are open, and allow specifying aggregated information, such as units (metrics) for a parameter value:

```
    parameter NODE_SIZE: float := 0.18;
    assign NODE_SIZE'unit = um;
```

Eventually, users define failure modes almost in the same way they define parameters. As instance, if a user wants to define a Single Bit Upset failure mode he may resort to the following snippet of code:

```
    fail_mode SBU;
    assign SBU'description = "Single bit upset" ;
    assign SBU'unit = FITS;
```

The *fail_mode* keyword helps distinguish between general parameters and failure modes, but the way attributes are defined is the same for *parameters*:

```
      assign SBU'rate = NUMBER_REGISTERS*FF_PER_REG/pow(2,20);
```

In this example, the rate of the SBU is a formula taking into account two (previously) defined *parameters*.

Environments and *Requirements* follow the same syntax of components. As an example, we propose the following "cold" environment:

```
environment COLD_COMPONENT_ENV;
    // Temperature
    parameter TEMPERATURE: float;
    assign TEMPERATURE'unit = C;
    assign TEMPERATURE'VALUE = 30;

    // Voltage
    parameter VOLTAGE : float;
    assign VOLTAGE'VALUE = 1.0;
endenvironment
```

In the environment we set two parameters: the temperature and the (reference) voltage. If needed, units can be defined as well. The concept of environment is particularly important in CLERECO to model the concept of operation mode defined in deliverable D2.3 (*Definition of operation modes for future systems*).

As an additional example, Figure 3 shows the basic usage of constants and parameters applied to the definition of a simple SRAM component.

```
01:component SIMPLE_SRAM;
02:
03:   // Parameter Declaration
04:   parameter VOLTAGE : float := 1.0;
05:   assign CORE_VOLTAGE'UNITS = VOLTS;
06:   parameter DIE_TEMP : float := 25.0;
07:   assign VOLTAGE'UNITS = CELSIUS;
08:
09:   // Parameter to be modified by user
10:   parameter NUM_BITS : integer := 1024*1024; //number of bits
11:
12:   // Constants specific to modeling this SRAM
13:   constant A_DIFF : float    := 3.2;
14:   constant Q_COL_EFF : float := 0.6;
15:   constant MBU_RATIO : float := 0.25;
16:
17:   // Define Radiation Induced Failure Modes
18:   fail_mode SBU;  // Rad. induced single bit error
19:   fail_mode MBU;  // Rad. induced multiple bit error (same
word)
20:   fail_mode SEFI; // Radiation induced failure of entire device
21:
22:   assign SBU'UNITS = FITS;
23:   assign MBU'UNITS = FITS;
24:   assign SEFI'UNITS = FITS;
25:
26:   // Equations to specify rate of defined failure modes
27:   assign SBU'RATE = NUM_BITS * A_DIFF * EXP( - CORE_VOLTAGE /
                        Q_COL_EFF );
28:   assign MBU'RATE = SBE'RATE * MBU_RATIO;
29:   assign SEFI'RATE = 10; // obtained from testing
30:
31:endcomponent SIMPLE_SRAM
```

Figure 3: RIIF description of a simple SRAM component

The following section will introduce the RIIF-2 extensions introduced in CLERECO

2.3.2. RIIF-2 Extensions

In RIIF, the possibility to define templates is limited to the definition of a hardware component in which common desired information items are listed without providing their values. While this mechanism is effective for small libraries of components, an explicit set of statements to define and manipulate templates is desirable to improve the robustness of RIIF descriptions in case of large libraries and to ease the implementation of automatic verification tools. In order to implement a full template mechanism (such as in most high-level programming languages), RIIF-2 introduces a new **template** statement. A template enables one to define a set of constants, parameters and failure modes that must be defined in all components implementing the template. The example of Figure 4 defines a template for an SRAM (lines 1-24) and one for a flip-chip package (lines 33-44). Predefined values for parameters and constants can be defined directly in the template as for instance PACKAGE_TEMP'UNITS (line 39). In a template, predefined values can be assigned either inline or through the new introduced keyword **impose**. The value of predefined parameters and constants does not need to be reassigned in those components implementing the template. Undefined values can be defined through the use of the **abstract** keyword. Within a template, each definition identified with the **abstract**

keyword simply includes a label and a data type as for instance the definition of the CORE_VOLTAGE parameter (lines 4-7). Abstract parameters identify *mandatory* information that must be defined in all components implementing the template. Once a template is applied to a component, the user is required to define the actual values for all abstract items described within the template. The application of a template to a component is described through the new keyword **implements** as for example at line 46 where a flip-chip SRAM is defined. Multiple templates can be implemented by the same components, thus allowing complex usages when a complex hierarchy of components must be described. In our example the defined component implements both the SRAM and the package template.

```
01:template SRAM_TEMPLATE;
02:
03:  // All SRAMs must voltage, temperature and size information
04:  abstract constant          NAME : string;
05:  abstract constant  MANUFACTURER : string;
06:  abstract parameter CORE_VOLTAGE : float;
07:  abstract parameter     NUM_BITS : integer;
08:
09:  // All SRAMs must have radiation induced failure modes
10:  fail_mode RAD_FM[];
11:  // All SRAMs must have permanent failure modes
12:  fail_mode PER_FM[];
13:
14:  abstract        RAD_FM[SBU]'RATE;  // single bit upset
15:  impose          RAD_FM[SBU]'UNITS  = FITS;
16:  abstract        RAD_FM[MBU]'RATE;  // multiple bit upset
17:  impose          RAD_FM[MBU]'UNITS  = FITS;
18:  abstract        RAD_FM[SEFI]'RATE;  // control logic errors
19:  impose          RAD_FM[SEFI]'UNITS = FITS;
20:  abstract        RAD_FM[SEL]'RATE;  // single event latchup
21:  impose          RAD_FM[SEL]'UNITS = FITS;
22:  abstract        PER_FM[SSAF]'RATE;  // single stuck-at
fault
23:  impose          PER_FM[SSAF]'UNITS = FITS;
24:endtemplate
25:
26:template SYNCSRAM_TEMPLATE extends SRAM_TEMPLATE;
27:  //All synchronous SRAM must specify the clock speed.
28:  abstract parameter     CLK_Speed      : integer;
29:  impose           CLK_Speed'UNITS= MHZ;
30:   .....
31:endtemplate
32:
```

```
33:template FLIP_CHIP_TEMPLATE;
34:
35:// All flip-chip packages must contain the following info.
36: abstract constant                NAME : string;
37: abstract parameter           NUM_BUMPS : integer;
38: abstract parameter        PACKAGE_TEMP : float;
39: impose          PACKAGE_TEMP'UNITS = C;
40:
41: // All Flip-Chip packages have these failure mechanisms
42: abstract fail_mode        OPEN_BUMP;
43: abstract fail_mode        DIE_CRACK;
44:endtemplate
45:
46:component CY7C1263XV18 implements
SYNCSRAM_TEMPLATE,FLIP_CHIP_TEMPLATE;
47:
48: set SYNCSRAM_TEMPLATE.NAME  = "CY7C1263VX18";
49: set MANUFACTURER            = "CYPRESS";
50: set CORE_VOLTAGE            = 1.8;
51: set NUM_BITS                = 37748736;   // 36 Mbit
52: set CLK_Speed               = 633;
53: set FLIP_CHIP_TEMPLATE.NAME = "165-LBGA";
54: set NUM_BUMPS             = 165;
55: set PACKAGE_TEMP'MIN      = 0;
56: set PACKAGE_TEMP'MAX      = 70;

57: set RAD_FM[SBU]'RATE      = .....;
58: set PER_FM[SSAF]'RATE       = .....;
59: endcomponent
```

Figure 4: RIIF-2 description of a flip-chip synchronous SRAM.

The expression power of the improved template mechanism is further increased when coupled with the introduction of the *inheritance* capability of RIIF-2. Inheritance is described by redefining the use of the **extends** keyword used in RIIF to denote the implementation of components from templates.

Through inheritance, both templates and components can be redefined, thus creating new templates or components that inherit all the definitions contained in their parent and modify only those portions that differ. Lines 26-31 of Figure 4 define a synchronous SRAM template that extends the basic SRAM definition. This refined template defines the SRAM clock frequency as an additional parameter required to characterize the component. Together with the **extends** keyword the new keyword **self** is used whenever a child template/component needs to redefine the value of a constant/parameter based on the value of the same constant/parameter defined in the parent template/component. An example of usage of this mechanism is presented subsequently in Figure 5 later in this section.

Further language extensions proposed in RIIF-2 focus on the introduction of *complex data structures*. The RIIF language only supports the definition of fixed size numerically indexed arrays. However, several cases do exist in which information must be associated to a set of labels to make it easy for retrieval during automated system reliability analysis. For this reason, the RIIF-2 language includes a new *associative array* data type. An example of an associative array is the definition of the SRAM failure modes in Figure 4. In order to group them into radiation induced failure modes and permanent failure modes they are defined through two associative arrays (RAD_FM and PER_FM at lines 10 and 12). The empty brackets are used to denote the associative arrays. In particular they indicate that the number of elements is undefined and new elements can be freely appended to the array. The index of each element is defined

when the element is created: <VECTOR_NAME>[<element_label>] = <value>. In this way, there is no need to number the vector elements and the access is based on the label used as an index.

Finally, the extended RIIF language introduces a new data type: **table**. Tables are the perfect data structure whenever groups of heterogeneous information must be aggregated together to maintain their informative content. The definition of a table includes the definition of a header defining the columns of the table and the definition of the table content. An example of its use is provided in Figure 5 and Figure 6 later in the document. Together with the table data type a new operator denoted with the symbol **[#]** is introduced. When applied to a table column it denotes an iterative access to all rows of the table. It is particularly useful whenever the value of a parameter must be expressed as a function of values contained in a table.

The proposed RIIF-2 extensions can be efficiently used to describe software componentts. Software components are in general difficult to profile. They can be characterized statically (i.e., without execution), or dynamically (i.e., collecting run-time information). Dynamic properties are particularly difficult to collect since they are strongly related to the input data sets.

We show here how the expression power of RIIF-2 can be efficiently used to model and manage reliability related information for software components, thus addressing the full system stack. We start from the set of software parameters described in Table 6, Table 7 and Table 8.

One of the main information to describe when dealing with software components is that hardware level failure modes (e.g., SBU) may deviate the correct software execution generating a set of possible *software faulty behaviors* (SFBs). As reported before we model this through the definition of a set of *software fault models* (SFMs). Each SFM translates the effect of a hardware failure model into the software domain (e.g., an SBU translates into a wrong data of an instruction). SFMs represent the link between the hardware and the software layer of a full system stack. The SFBs describe how a software component reacts to a given SFM.

Figure 5 shows an example of how RIIF-2 can be used to model a system including hardware and software. For brevity, the model of the full hardware layer is reduced to a single hardware component VECTORCALC_CORE able to execute vector computations. In this component we assume that fault injection experiments have been used to measure the occurrence rate of a set of SFMs in the presence of hardware failure modes and this information has been modeled by the SW_FM table (lines 5-9). Lines 12-32 define a high-level SW_COMPONENT template modeling the above-mentioned basic information items characterizing a software module. It uses the extended RIIF-2 template formalism. In this template the constant SFB_ITEMS defines a set of labels that identify 11 SFBs expressing the time properties of the module (IN_TIME, UNDETECTABLE, EARLY, LATE), the responsiveness of the module (FULL_UNRESPONSIVE, PARTIAL_UNRESPONSIVE, RESPONSIVE) and the data integrity of the module (DATA_BENIGN, NO_DATA, EDC, NON-EDC). Egregious Data Corruptions (EDCs) indicate software outcomes that significantly deviate from the error-free outcomes while NON-EDC indicate small or no deviations in the obtained results.

Line 20 introduces a key information for the template. Each instance of SW_COMPONENT must define the target hardware execution platform through the new RIIF-2 keyword **platform** thus establishing a link between the software and the hardware layer as explained later in this section. Lines 23-30 show instead an example of the use of the newly introduced table data structure to describe both time constraints and SFBs. Both items cannot be described as simple single-value parameters, but require an aggregation of a set of heterogeneous information. In particular the template declares the two tables and their headers that in turn define the information that will be stored when the template will be implemented by a component. A table header is a vector whose elements are defined inline with a comma separated list enclosed within {} brackets. The number of elements of the vector sets the header dimension dynamically.

To further emphasize on the capability of the table data structure, let us consider the implementation of the SW_COMPONENT template reported in lines 34-54 of Figure 5. In this component besides setting the basic information defined in the template line 43 sets the value of the target platform and links this software to the VECTORCALC_CORE. Through the SW_FM table of the VECTORCALC_CORE component that links its failure modes to the SFMs, and through the SFB table of the VADD component that links each SFM to the SFB reliability information can be efficiently propagated from the hardware to the software layer of the stack. Moreover, line 44 shows how the RIIF **child_component** keyword already available in the initial version of the language can be used to model the software hierarchy.

In the VADD component, the use of the new table data type can be appreciated. Lines 46-48 define the items (rows) of the TIME_CONSTRAINTS table. Each item reports the execution time (EXEC_TIME column), the average and the maximum time (AVG_TIME, MAX_TIME columns) for a given workload (WORKLOAD column). An even more complex use of the table data type is used in lines 48-52 to define the items of the SFB table. In this case, each row of the table identifies an SFM. The probabilities of occurrence of each SFB in the presence of the SFM are defined. This is accomplished through the two columns named OCCURRING_SFB representing the list of SFBs and OCCURRING_SFB_RATE representing the associated probabilities. These two elements are two arrays defined within a column of a table. Resorting to the flexibility of these new data structures we have been able to represent complex data in a very compact and expressive manner.

Figure 6 (lines 1-12) shows instead an example of how inheritance can be easily used to define a modified version of the same software application implementing a software error detection mechanism based on variable duplication. In this example we show how the **self** operator can be used to easily model the time overhead introduced by the inserted protection mechanism w.r.t. the timing of the original application. For the sake of readability, the definition of the improved masking probabilities is not reported.

Finally, Figure 6 (lines 13-20) shows another way to describe the component proposed in Figure 6. In this case we use the iterative operator [#] introduced in RIIF-2 to redefine all entries of the TIME_CONSTRAINTS table with a single definition. The operator applies the same formula to all entries of a given table. This feature is particularly useful when managing big tables whose lines must be processed according to the same criteria, for example, scaling of all failure rates.

```
01:component VECTORCALC_CORE;
02:  // An hardware component performing vectorial calculations
  ...
03:  parameter fail_mode SBU;
04:  assign SBU'RATE = 10; //obtained from radiation tests
  ...
05:  parameter SW_FM: table;
06:  assign SW_FM'HEADERS = {FAILMODE, SFM, RATE};
07:  assign SW_FM'ITEMS = { // Obtained from fault injection
08: [ "SBU", "WRONG_DATA", 0.3 * SBU'RATE ],
09: [ "SBU", "WRONG_INSTRUCTION", 0.2 * SBU'RATE ], ... };
10:endcomponent
11:
12:template SW_COMPONENT;
13:  // All programs must define the name, size, …
14:  abstract parameter         NAME : string;
15:  abstract parameter         SIZE : integer;
16:  abstract parameter        LOOPS : integer ;
17:  abstract parameter   PROTECTION : enum {NONE, VAR_DUP, ...};
18:  abstract parameter  READ_ACCESS : integer ;
19:  abstract parameter WRITE_ACCESS : integer ;
20:  abstract platform   executed_on;

21:  // List of possible SFB considered in our library
22:  abstract constant    SFB_LIST:= {IN_TIME,DETECTABLE, EARLY,
     LATE, FULL_UNRESPONSIVE, PARTIAL_UNRESPONSIVE, RESPONSIVE,
     DATA_BENIGN, NO_DATA,EDC, NON-EDC};
24:
25:  // Timing constraints depending on the workload
26:  abstract parameter TIMING_CONSTRAINTS   : table;
27:  impose             TIMING_CONSTRAINTS'HEADERS = {WORKLOAD,
                        EXEC_TIM, MAX_TIME, AVG_TIME};
28:
29:  // Software faulty behaviors table defining the probability
     of occurrence of each SFB given the occurrence of each SFM.
30:  abstract parameter SFB             : table;
31:  impose             SFB'HEADERS = {SFM_TYPE, SFM,
                        OCCURING_SFB, OCCURRING_SFB_RATE};
32:endtemplate
33:
```

```
34:component VADD implements SW_COMPONENT;
35:  set NAME                  = "Vector ADD";
36:  set SIZE              = 524;
37:  set SIZE'UNITS            = instructions;
38:  set PROTECTION            = NONE;
39:  constant NUMBER_OF_ITEMS := 10000;
40:  set READ_ACCESS           = 76 * NUMBER_OF_ITEMS / 10000;
41:  set WRITE_ACCESS          = 75 * NUMBER_OF_ITEMS / 10000;
42:  set LOOPS                 = 3;
43:  set executed_on           = VECTORCALC_CORE;
44:  child_component   VPRINT;
45:  .....
46:  assign TIMING_CONSTRAINTS'ITEMS = {
47:    [ TEST_BENCH1, 0.0000001, 2, 0.000001 ],
48:    [ TEST_BENCH2, 0.0000003, 2.1, 0.0000004 ] };
49:  assign SFB'ITEMS = {
50:    [ "permanent", "WRONG_DATA", SFB_ITEMS, { 0.893, 0.107, 0,
       0, 0.891, 0.42, 0.67, 0.413, 0.109, 0.052, 0.426 } ],
51:    [ "permanent", "INSTR_REPLACEMENT", SFB_ITEMS, { 0.274,
       0.726, 0, 0, 0.378, 0.348, 0.274, 0, 0.726, 0, 0.274 } ],
52:    [ "transient", "WRONG_DATA", SFB_ITEMS, { 0.893, 0.009, 0,
       0.098, 0.987, 0.001, 0.012, 0.968, 0.013, 0, 0.019 } ],
53:    [ "transient", "INSTR_REPLACEMENT", SFB_ITEMS, { 0.614,
       0.309, 0, 0.077, 0.309, 0, 0.691, 0.691, 0.309, 0, 0 } ]};
54:endcomponent
```

Figure 5: RIIF-2 System modeling with both hardware and software components

```
01:component VADD_VARIABLE_DUPLICATION_VER1 extends VADD;
02:  set NAME = "Vector ADD with Variable Duplication";
03:  set PROTECTION = VAR_DUP;
04:  assign TIMING_CONSTRAINS'ITEMS = {
05:   (PROTECTION == VAR_DUP)?
06:    [ "TEST_BENCH1", self+0.001, self+0.2, self+0.0015] :
07:   [ "TEST_BENCH1", self , self , self ],
08:   (PROTECTION == VAR_DUP)?
09:   [ "TEST_BENCH2", self+0.001, self+0.2, self+0.0015] :
10:   [ "TEST_BENCH2", self , self , self ] };
11:endcomponent
12:
13:component VADD_VARIABLE_DUPLICATION_VER2 extends VADD;
  ...
14:  assign TIMING_CONSTRAINS'ITEMS[#][EXEC_TIME]=
15:   (PROTECTION == VAR_DUP)? self + 0.001 : self ;
16:  assign TIMING_CONSTRAINS'ITEMS [#][MAX_TIME]=
17:   (PROTECTION == VAR_DUP)? self + 0.2 : self ;
18:  assign TIMING_CONSTRAINS'ITEMS[#][AVG_TIME]=
19:   (PROTECTION == VAR_DUP)? self + 0.0015 : self ; };
  ...
20:endcomponent
```

Figure 6: RIIF-2 Vector_ADD software with protection mechanisms.

2.3.3. Implementation

A prototype C++ parser able to read and process RIIF-2 models has been developed as part of the CLERECO-toolsuite. It integrates the basic Antlrr3[7] specification of the RIIF grammar with the new constructs and data types.

3. System Description Language

Differently from the internal characterization of a component, the formalism used to describe the system architecture is strictly related to the model introduced in CLERECO to perform system level reliability estimations. In deliverable **D5.2.2** we introduced the CLERECO system reliability model. It is a *Component-Based* (CB) reliability model. In CB reliability modeling, reliability at system level is estimated using reliability information and other properties (e.g., size, complexity, etc.) of its individual components that are described using the RIIF-2 language and their interconnections, the system architecture, whose description language is introduced in this section.

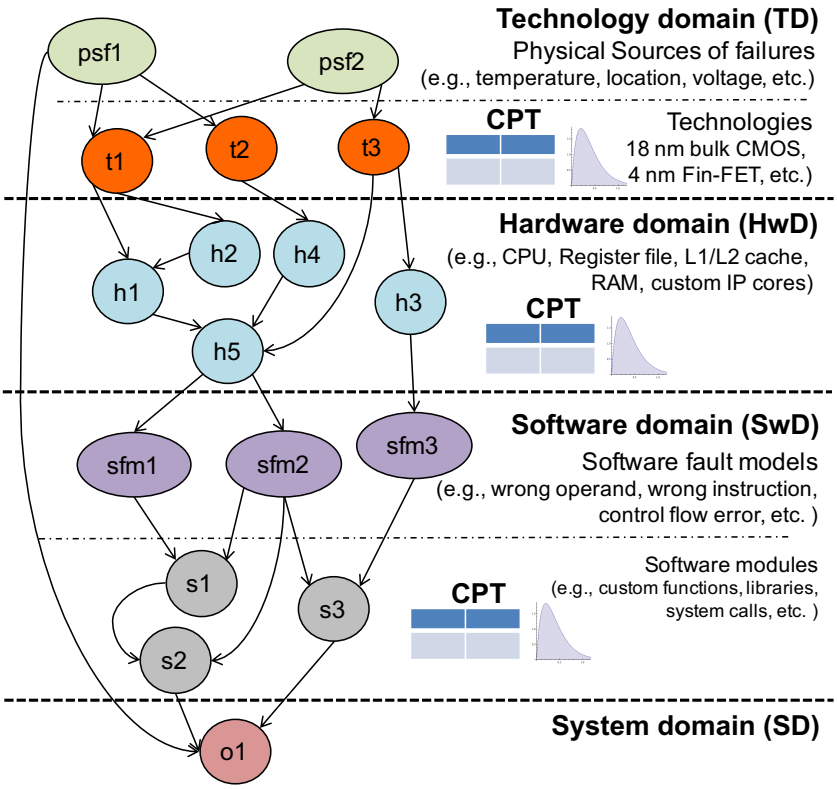


Figure 7: The Bayesian model of a system. The system is split into four domains: technology domain (TD), hardware domain (HwD), software domain (SwD) and system domain (SD). The topology of the network provides the qualitative description of the system. Conditional probability tables (CPT) associated to each node of the network provide the quantitative description of the system.

[7] *http://www.antlr.org*

Figure 7 summarizes the idea of the system reliability model introduced in **D5.2.2.** The model is a Bayesian model in which a full system is described in terms of a Bayesian Networks (BNs), i.e., a direct graph. Each node represents a component or subcomponent of the system, while arcs define paths in which errors can propagate. Nodes are hierarchically organized into a set of four domains (TD, HwD, SwD and SD) directly mapped to the different layers composing the system. The domain hierarchy follows the error propagation flow within the system.

Each node must be characterized with information expressed in the form of conditional probabilities that quantitatively describe how errors propagate from one component to the other component.

Bayesian Networks are exploited in several application domains. Therefore, description languages to formally describe the structure of a BN and its quantitative model (i.e., the set of conditional probabilities characterizing each node) are already available:

- XDSL File Format is a open XML Domain Specific Language developed for the description of Bayesian networks.
- DSL File Format is a open Domain Specific Language developed for the description of Bayesian networks.
- Ergo File Format: it is a simple file format for Bayesian networks used by Noetic Inc. in their program Ergo, Version 1.0. The format has become popular in the Uncertainty in Artificial Intelligence community because several example networks were saved in this format and shared among various researchers. The format includes only node identifiers, state names, conditional probability tables, and locations of the node centers. No other information can be included. Moreover Noetic Inc. has now changed its file format for new releases of the software and the old format is no longer supported.
- Netica File Format: it is a closed format used by Norsys Inc.
- BN interchange Format: this format is an attempt to design a common format for graphical probabilistic models. The format has not yet been established as a standard and different implementations in different tools do exist.
- Hugin File Format. This is a close format used by Hugin A.G. in their program Hugin.
- KI File Format: a format used by Knowledge Industries (www.kic.com), Inc., in their program DExpress.

Unfortunately, a standard for the representation of BNs has not been yet established and the different file formats have been created in order to work with several software and APIs.

Among the available option in CLERECO we focus on open language definition that are can be freely used and integrated and commercialized in our tools. This limits our decision to the three options: XDSL, DSL and BN interchange Format. Among them our choice was to base our system descriptions on a modified version of the XDSL language. The main reason for choosing the XDSL language is that it is based on a XML scheme and generic open-source XML parsers are available for several programming languages and can be easily extended, modified and integrated in the CLERECO tools.

In particular, we improved the XDSL language to include the following features:

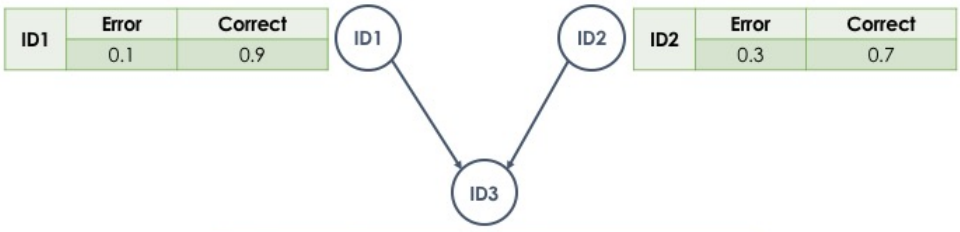
- The system description must include a way to link each component to its RIIF-2 description containing reliability related information on which the conditional probability table of the node is compiled.
- A mechanism to cluster nodes in order to express hierarchical relations among components in a system (i.e., a microprocessor and its sub components).
- A mechanism to describe extended architectural constraints among the node of the network. This constraint is particularly important in order to support the system optimization heuristic described in deliverable D5.3 that require to explore the design space by modifying the architecture of the system (i.e., the topology on the network) according to user defined design constraints.

3.1. XDSL basic definitions

The SMILE (Structural Modeling Inference, and Learning Engine) is a fully platform independent library of functions implementing graphical probabilistic and decision-theoretic models, such as Bayesian networks, influence diagrams, and structural equation models. It has been developed at Decision Systems Laboratory in Pittsburg. Its individual functions, defined in SMILE Application Programmer Interface, allow to create, edit, save, and load graphical models, and use them for probabilistic reasoning and decision making under uncertainty. SMILE supports directly object oriented methodology. SMILE is implemented in C++ and is platform independent. Individual classes of SMILE are accessible from C++ or (as functions) from C programming languages.

CLERECO exploits the SMILE library as a core function to perform Bayesian network reasoning. SMILE fully supports the CDSL format targeted in the project to describe the architecture of the system in terms of BNs.

Figure 8 shows a simple example of a generic BN that will be used in the remaining of this section to shortly summarize the XDSL description. It is a very simple network, including only three nodes. Each node has two states: Error or Correct. Nodes ID1 and ID2 are associated to probability tables that define the probability of being in one of the two states. For node ID3, the probability table is a Conditional Probability Table (CPT). According to BN theory, The CPT describes the probability of each state of ID3 with respect to all the combinations of state of the parent nodes (ID1 and ID2).



ID1	Error	Correct
	0.1	0.9

ID2	Error	Correct
	0.3	0.7

	ID1	Error		Correct	
	ID2	Error	Correct	Error	Correct
ID3	Error	0.99	0.87	0.89	0.02
	Correct	0.01	0.13	0.11	0.98

Figure 8 - A generic BN

The XDSL format describes the network as a collection of nodes identified through the use od the `nodes` tag.

```
<nodes>
  ...
</nodes>
```

Each node as associated a set of basic information:

- A textual unique identifier of the node.
- The set of states defined for the node (see D5.2.2 to further details)
- The probabilities associated to the states of the node.
- The list of parents of the node. This describes the topology of the network.

These information items are described in XML through the generic `cpt` tag that defines a Conditional Probability Table:

```
<cpt id="<textual node ID>"
    <state id="<State ID #1>" />
    ...
    <state id="<State ID #n>" />
    <parents>...</parents>
    <probabilities>...</probabilities>
</cpt>
```

Each `state` is defined through a unique ID, whereas `parents` is a list of space-separated node IDs. The same applies for the conditional probabilities that are described as a space-separated list.

The network of Figure 8, corresponds to the following XDSL code:

```
<nodes>
  <cpt id="ID1">
    <state id="Error" />
    <state id="Correct" />
    <probabilities>0.1 0.9</probabilities>
  </cpt>
  <cpt id="ID2">
    <state id="Error" />
    <state id="Correct" />
    <probabilities>0.3 0.7</probabilities>
  </cpt>
  <cpt id="ID3">
    <state id="Error" />
    <state id="Correct" />
    <parents>ID1 ID2</parents>
    <probabilities>0.99 0.01 0.87 0.13 0.89 0.11 0.02
0.98</probabilities>
  </cpt>
</nodes>
```

One of the main limitations of the XDSL format is that probabilities are expressed in `probabilities` tag based on the order of the parent nodes reported `parents` tag. This is a major limitation for the application of this format in the CLERECO reliability model and in particular in the optimization heuristics described in deliverable D5.3. In fact during system design exploration the topology of the network is often modified to find the best architecture maximizing the reliability of the system. Next sections will address this problem introducing an extended format that better suits the reliability modeling developed in the project.

3.2. XDSL CLERECO meta-information schema

The XDLS format introduced in the previous section is a very effective format to formally describe the basic information required to build a generic BN. This information is the minimum required by generic BN solvers to perform Bayesian reasoning on the network. In this section we want to extend this format adding additional information required by the CLERECO system level analysis algorithms to perform reliability analysis.

When working with XML based languages, two different approaches can be followed to extend a description schema:

1. Defining a new set of custom tags able to model the required information,
2. Using general tags whose meaning is modified resorting to the attributes of the tag.

The first approach has the advantage of adding expression power to the language but, at the same time, it requires introducing significant changes in the parsers in order to recognize the new tags. This is a major problem especially when revisions or extensions of the language are introduced to deal with new requirements.

The second approach is instead more generic. It makes human reading of the produced files less intuitive but, at the same time, requires minimum modifications to the parsers and enables easy extensions of the language.

Given the need of having a very generic language, easy to modify based on the new requirements that we faced during the development of our tools and models we decided to opt for the second extension mode. Extra information are included in the description of the model resorting to the use of generic `property` tags to enrich the description of a node inside the `cpt` tag. A general definition of a `property` is as follows:

```
<property id="<property unique ID">property value</property>
```

By using several `property` tag, we are able to add any type of information to each node.

This solves one of the first requirements of our system description format, that is the possibility of directly introducing in the model component information described in the RIIF-2 description of the competent. In order to keep the model light, the number of RIIF-2 information migrated directly in the model must be reduced at a minimum. At the same time a property defining the filename containing the full RIIF-2 description of the component must be introduced in the definition of each node. This establish a link between the system model and the description of the component that can be browsed by the developed tools in order to have access to all required information when needed.

The second main extension required in our system description is the possibility of clustering different nodes of the system into macro-components. This is required during design exploration to enable automatic modification of the system architecture (e.g., replacing a set of nodes modeling the architecture of a given microprocessor with another set of node modeling a different architecture). We therefore introduce the concept of cluster in a node. A `Cluster` is a set of nodes belonging to the same macro-component. Practically, by introducing the cluster property in the definition of each node as follows

```
<property id="Cluster">Cluster label</property>
```

we can easily group set of nodes.

However, the simple introduction of the cluster property does not completely solve the problem of enabling design exploration algorithms to automatically replace a macro-component with another macro-component. In fact, when a cluster of nodes in unplugged from the model and a new one is introduced, the nodes of the new cluster must be properly connected to the other nodes of the system.

To better understand this problem let us consider the system depicted in Figure 9 that shows a small BN representing, on the left side, the architecture of a simplified version of x86 microprocessor connected to a main memory (RAM) and its replacement with the model of a simplified version of an ARM microprocessor, on the right side. Each node is annotated with the cluster information in purple. The blue labels in the center of the node represent the ID of the node. The reader may notice that, when the x86 cluster is removed and the ARM cluster is introduced, the RAM node requires to be correctly connected to some of the nodes of the new cluster. In this case the RAM must be connected to the L2 cache of the microprocessor.

In order to cope with this problem we need to introduce additional meta-information in the model to *virtually* define the role of each node in the system. Red labels Figure 9 represent this extra information. They define classes of equivalence among nodes in different clusters.

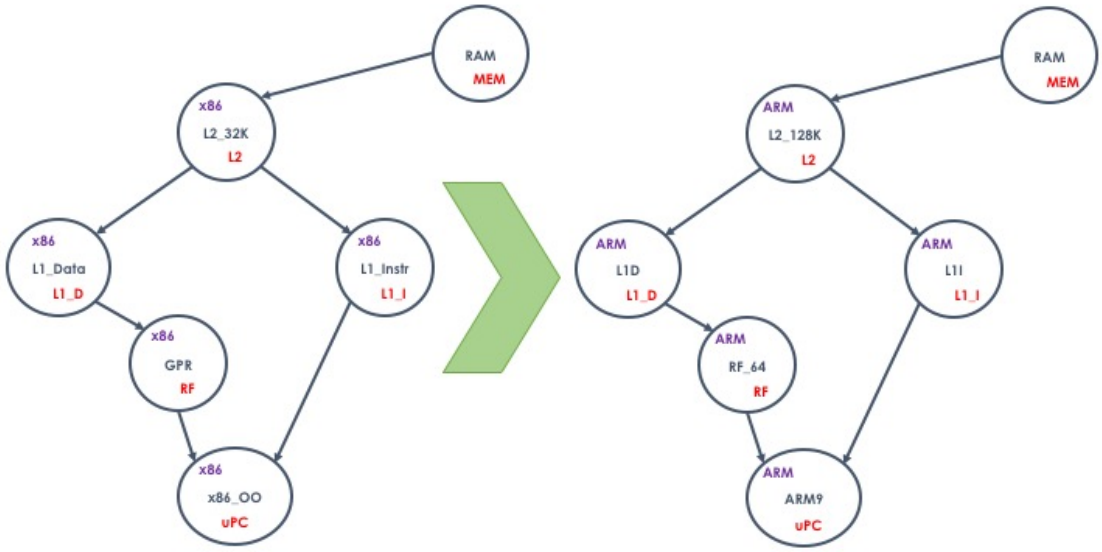


Figure 9 - A macro-component replacement with meta-information annotated for each node.

This is translated in the description by introducing an additional property for each node as follows:

```
<property id="VID"> [value] </property>
```

We call the property *Virtual ID* (`VID`) because it represents a unique identifier across similar clusters of node.

Both `Cluster` and `VID` can be omitted if not necessary. In the example of Figure 9 the main memory `RAM` has a `VID` property but, being a simple element it does not required the definition of the Cluster property.

The `VID` property has another important role in the description of the system. As explained before in the original version of the XDSL schema, the definition of the parents defines a fixed orders used to define the orders in which probabilities are represented in the probability tag. This is effective when the structure of the network is fixed but creates problems when cluster replacements are performed since this may change the order of the parents.

To cope with this problem we exploit the VID property to create two different parents ordering in a node. The XDSL parents tag is used to define the structure of the network as in the standard XDSL description, and the order of the parents can be freerly modified during the replacement process. At the same time a new parents property is defined as follows:

```
<property id="parents"> [VID of all parents] </property>
```

This property defines the order of the parents not based on the parentis but based on the VID. Since the VID is not related to a specific node but is related to a role in a cluster, this ordering is persistent among different clusters. Probabilities are then defined according to this ordering that remains fixed among clusters instead of the XDSL parent ordering, thus enabling to efficiently implement the replacement process.

With the proposed extensions to the original XDSL format we are able to efficiently define the architecture of a complex digital system including all meta-information required to support the CLERECO tools described in deliverables **D5.2.2** and **D5.3**.

It is important to remark here that, by the use of the generic property tag we have been able to keep full syntactic compatibility with the original XDSL format. This is particularly important since the SMILE library is still able to properly parse the CLERECO extended XDSL models thus allowing to save resources in the development of the parsers required to integrate this de-

scription format in the developed tools.

3.3. The SMILE Wrapper Implementation

The SMILE library is public available library that comes without sources. It is one of the most used libraries, due to its performance and the large set of implemented BN solving algorithm. For this reason, we rely on this library for all statistical CLERECO tool. However, the library has a set of limitations that prevent its directly use in our developed tools, and in particular it does not support the XDSL extensions introduced in this document.

To overcome these limitations, we developed a full C++ wrapper based on the QT Framework. The wrapper allows to quickly implement all previously discussed enhancement, together with bridge functions to connect the BN generation to the RIIF-2 description of single components. Moreover, it introduces some new capabilities that SMILE does not provide:

1. Network replacement. SMILE only provides simple methods to add or remove single nodes and edges. The wrapper supports dynamic reconfiguration / replacements of clusters of nodes resorting to Clust and VID properties defined for each node.
2. Simplified methods for data access. The wrapper includes methods to access all properties of a node by their keys (strings) in place of the indexed access provided by SMILE. Moreover, it also supports automated full data recovery and store that is not present in the library (thus all properties can be stored or retrieved at once).
3. Full parents' management. Together with the network replacement the wrapper provides methods to add, remove or change parents of a node also considering the coherency of the CPT. In fact, the SMILE library automatically modifies the nodes CPT when one of its parent is removed or added (in this cases the CPT is reduced or expanded without relating with the parents so that probabilities could be wrong). Instead all methods to change parents order do not change the CPT order. The wrapper fully supports all this operations.

To conclude, the introduction of the wrapper also decouples the developed tools from the SMILE library itself. If in the future different libraries supporting optimize BN solvers will be available, they can be easily plugged in the CLERECO tools by sampling updating the wrapper to support the new libraries, without any modification to the remaining parts of the tools. As an example, a GPGPU based library implementing accelerated solvers for BNs is under development by partner POLITO. Once ready, this library will be easily exploitable by the CLERECO tools.

4. Conclusion

This deliverable represents the steps performed in CLERECO toward the definition of well-defined formalism to describe reliability information of a full system. In particular, the document covers the definition of a formalism to describe reliability information of hardware and software components of a system through the new RIIF-2 format, and a formalism to describe the architecture of the system through the extended XDSL format. The defined languages represent the first building block for all system level reliability analysis tools developed in CLERECO.

6. Bibliography

[1] Reliability Modeling and Prediction, ser. Military standard. U.S. Department of Defense, 1981. [Online]. Available: https://books.google.ca/books?id=b50QAQAAMAAJ

[2] M.Modarres, M.Kaminskiy, and V.Krivtsov, Reliability Engineering and Risk Analysis: A Practical Guide. Taylor & Francis, 1999. [Online]. Available: https://books.google.ca/books?id=IZ5VKc-Y4_4C

[3] George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. Algorithm-based fault tolerance applied to high performance computing. Journal of Parallel and Distributed Computing, 69(4):410–416, 2009.

[4] Ramtilak Vemu and Jacob A Abraham. CEDA: Control-flow error detection through assertions. In International On-Line Testing Symposium (IOLTS), 2006.

[5] Melvin A Breuer, Sandeep K Gupta, and T.M. Mak. Defect and error tolerance in the presence of massive numbers of defects. IEEE Design & Test of Computers, 21(3):216–227, 2004.

[6] Evans, A; Nicolaidis, M.; Shi-Jie Wen; Alexandrescu, D.; Costenaro, E., "RIIF - Reliability information interchange format," IEEE 18th International On-Line Testing Symposium (IOLTS), 2012, vol., no., pp.103-108, 27-29 June 2012, doi: 10.1109/IOLTS.2012.6313849

[7] Schlichtmann, U.; Kleeberger, V.B.; Abraham, J.A; Evans, A; Gimmler-Dumon, C.; Glas, M.; Herkersdorf, A; Nassif, S.R.; Wehn, N., "Connecting different worlds — Technology abstraction for reliability-aware design and Test," Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014, vol., no., pp.1,8, 24-28 March 2014, doi: 10.7873/DATE.2014.265

[8] Argyrides, C.; Chipana, R.; Vargas, F.; Pradhan, D.K., "Reliability Analysis of H-Tree Random Access Memories Implemented With Built in Current Sensors and Parity Codes for Multiple Bit Upset Correction," IEEE Transactions on Reliability, vol.60, no.3, pp.528,537, Sept. 2011, doi: 10.1109/TR.2011.2161131

[9] Aliee, H.; Zarandi, H.R., "A Fast and Accurate Fault Tree Analysis Based on Stochastic Logic Implemented on Field-Programmable Gate Arrays," IEEE Transactions on Reliability, vol.62, no.1, pp.13-22, March 2013, doi: 10.1109/TR.2012.2221012

[10] Seifert, N., "Soft Error Rates of Hardened Sequentials utilizing Local Redundancy," 14th IEEE International On-Line Testing Symposium, 2008. IOLTS '08., vol., no., pp.49,50, 7-9 July 2008, doi: 10.1109/IOLTS.2008.61

[11] Bidokhti, N., "SEU concept to reality (allocation, prediction, mitigation)," Reliability and Maintainability Symposium (RAMS), 2010 Proceedings - Annual, vol., no., pp.1-5, 25-28 Jan. 2010, doi: 10.1109/RAMS.2010.5448078

[12] Mitra, Subhasish; Sanda, Pia; Seifert, Norbert, "Soft Errors: Technology Trends, System Effects, and Protection Techniques," 13th IEEE International On-Line Testing Symposium, 2007. IOLTS 07., vol., no., pp.4,4, 8-11 July 2007, doi: 10.1109/IOLTS.2007.61

[13] Clifton A. Ericson II, "Fault Tree Analysis - A History," Proceedings of the 17th International System Safety Conference, 1999, pages 87-96.

[14] Anderson, R.T., Reliability Design Handbook, March 1976, Illinois Institute of Technology. Research Institute and Reliability Analysis Center (U.S.)

[15] W3C, XML 1.0 Specification, http://www.w3.org/TR/REC-xml

[16] Philip Fennell, "Extremes of XML", Presented at XML London 2013, June 15-16th, 2013. doi:10.14337/XMLLondon13.Fennell01.

[17] W3C, Document Object Model reference, http://www.w3.org/DOM

[18] ISO, UML Standard Part 1, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=32624

[19] IBM Corporation, UML Basics, http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell

[20] Modarres, Mohammad, Mark Kaminskiy, Vasiliy Krivtsov, Reliability Engineering and Risk Analysis, New York, NY: Marcel Decker, Inc., p. 198., ISBN 0-8247-2000-8

[21] U.S. Department of Defense, "Reliability Modeling and Prediction", Electronic Reliability Design Handbook. B., 1998. MIL–HDBK–338B.

[22] Salvatore Distefano, Antonio Puliafito, Dependability Evaluation with Dynamic Reliability Block Diagrams and Dynamic Fault Trees. IEEE Trans. Dependable Sec. Comput. 6(1): 4-17 (2009), http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4385723

[23] Schlichtmann, U.; Kleeberger, V.B.; Abraham, J.A; Evans, A; Gimmler-Dumon, C.; Glas, M.; Herkersdorf, A; Nassif, S.R.; Wehn, N., "Connecting different worlds — Technology abstraction for reliability-aware design and Test," Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014, vol., no., pp.1,8, 24-28 March 2014, doi: 10.7873/DATE.2014.265

[24] Adrian Evans and Oliver Bringmann. RIIF DATE 2013 Workshop: Towards Standards for Specifying and Modelling the Reliability of Complex Electronic Systems. http://riif-workshop.fzi.de, 2013.

[25] Alfonso Sanchez-Macian, Pedro Reviriego, and Juan Antonio Maestro. Modeling Reliability of Memories Protected with Error Correction Codes with RIIF. In RIIF DATE 2013 Workshop: Towards Standards for Specifying and Modelling the Reliability of Complex Electronic Systems, 2013.

[26] [Online] Java RiiF CLI Project Page: http://code.google.com/p/java-riif-cli/.

[27] [Online] OpenRISC 1200 Project Page: http://opencores.org/or1k/Main_Page

[28] [Online] Univerisity of Michingan, MiBench: http://www.eecs.umich.edu/mibench/

[29] M. Bagatin, S. Gerardin, A. Paccagnella, C. Andreani, G. Gorini, C.D. Frost, Temperature dependence of neutron-induced soft errors in SRAMs, Microelectronics Reliability, Volume 52, Issue 1, January 2012, Pages 289-293, ISSN 0026-2714, http://dx.doi.org/10.1016/j.microrel.2011.08.011.