Cross-Layer Early Reliability Evaluation for the Computing cOntinuum

Project Number: FP7-611404

D5.1.1 - Input parameters and system modeling formal representation (preliminary)

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 1.1 – 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.

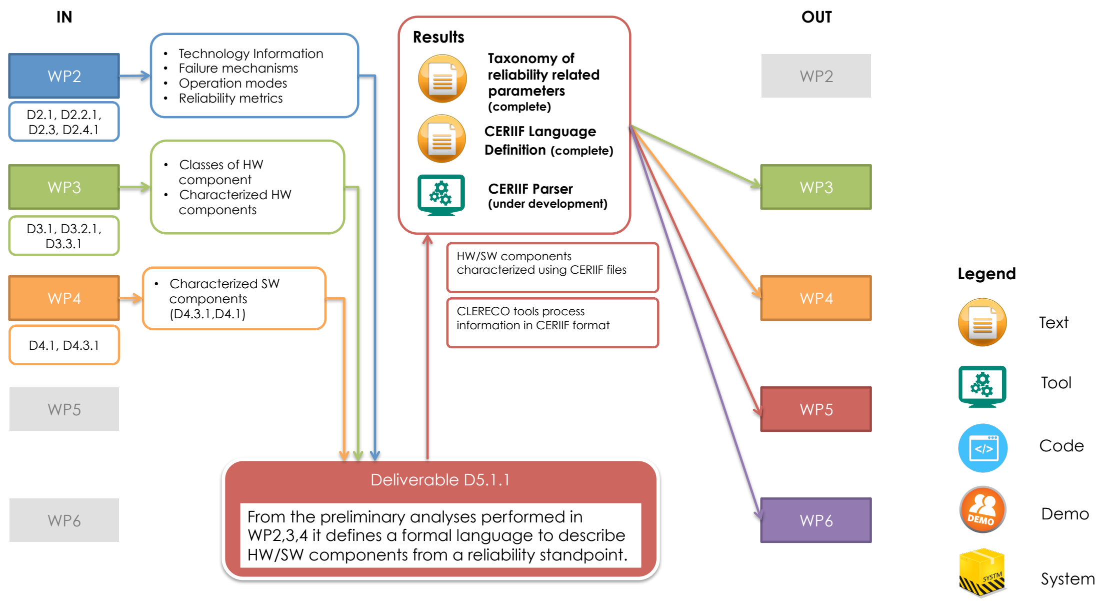
IN
WP2
D2.1, D2.2.1, D2.3, D2.4.1
WP3
D3.1, D3.2.1, D3.3.1
WP4
D4.1, D4.3.1
WP5
WP6
Technology Information
Failure mechanisms
Operation modes
Reliability metrics
Classes of HW component
Characterized HW components
Characterized SW components (D4.3.1,D4.1)
Results
Taxonomy of reliability related parameters (complete)
CERIIF Language Definition (complete)
CERIIF Parser (under development)
HW/SW components characterized using CERIIF files
CLERECO tools process information in CERIIF format
Deliverable D5.1.1
From the preliminary analyses performed in WP2,3,4 it defines a formal language to describe HW/SW components from a reliability standpoint.
OUT
WP2
WP3
WP4
WP5
WP6
Legend
Text
Tool
Code
Demo
System

Figure 1 - Inputs and Outputs of this Deliverable

D5.1.1 has two 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.

A library of characterized HW and SW components described using the CERIIF language defined in this deliverable is available as additional material to this document (see Table 11).

The document is organized in the following sections:

- **Introduction.** This section shortly overviews background research on reliability parameters and standardization of reliability related information.

- **Taxonomy of Components Reliability Parameters.** This section analyzes both hardware and software components to identify an initial set of useful parameters for their reliability characterization.
- **Description Language.** This section analyzes different choices for the identification of a language able to efficiently describe the system's components as expected from the previous section. It eventually identifies the target choice for the CLERECO project.
- **CLERECO Extended RIIF Language.** This section introduces the CLERECO Extended RIIF language that will be used in the project for component's description. It focuses on the language extensions that are introduced by CLERECO to the standard RIIF language. The extensions are useful to be able to describe all relevant parameters identified in this deliverable. The language overview is achieved by a set of relevant examples.
- **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, information about the reliability of an IC is essentially confined to the lower levels of the system stack (e.g., technology and circuit level). At this level, a deep understanding of the technology and of the implemented reliability mitigation techniques do exist. However, exposing this information to the higher levels of the system stack (e.g., architectural level, software level and system level) introduces several major challenges. 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.

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 [5]. 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.

Older publications focus on system's modeling for reliability analysis at the hardware layer [1][2][3]. 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 [6][7][8]. 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 [4] 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 language 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 language 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.

2. 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 overall system's description. The provided taxonomy is not meant to be exhaustive. It serves as a starting point for the definition of a dedicated description language. It will be continuously updated during the project.

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. Hardware Components Description

Table 1 summarizes the list of parameters identified within CLERECO 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. A detailed and accurate error rate information for each component represents the starting point to identify efficient reliability evaluation strategies. An error rate usually refers 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 in order 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 [25], 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. A complete description of this HW component, as well as additional components characterized in CLERECO is available Deliverable D3.3.1 (*Characterization of a set of hardware modules (preliminary)*).

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	mm²

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)
...	...	...	...

2.2. 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 within CLERECO for the characterization of a software component within a system, which is derived from the software characterization activities described in Deliverable D4.2.1 (*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 static or a dynamic analysis.	Number	-		NO
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.	Number			NO

	As for Cache misses, if available.				
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 Life-time	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 SW Fault Models and occurring SFBs must be

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

provided. They will help, at a first glance, to define among all SFBs the ones of interest during reliability estimation.

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 SFB Probabilities	The occurrence probability for each SFB	List	-	-	YES

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. The full characterization of this SW component, as well as additional components considered in CLERECO, are available in deliverable D4.3.1 (*Characterization of a set of software modules (preliminary)*).

Table 9 - Software Component Characterization example

Label	Data	Unit
Name	ADD Vector Application	-
Vendor	CLERECO	-
Type	SW	-
Class	Application	-
Subclass	Vector Operation Algorithm	-
Size	453	-
Reading Access Rate	76 * # of vector element / 10000	-
Writing Access Rate	75 * # of vector element / 10000	-
Memory Accesses	151 * # of vector element / 10000	-
Loops Number	3	-
Algorithm Complexity	N	-

Timing Constraints	Workload	Expected Execution Time	Maximum Accepted Execution Time	Average Accepted Execution Time
	Test Bench #1	10^{-6}	2	10^{-5}
	…			

Software Faulty Behaviors	Fault Type	Fault Model	Occurring SFB	Occurring SFB

[6] *For further details refers to Deliverable D4.2.1 (Software Characterization Methods)*

			Probabilities
Permanent	Wrong Data	In-Time, Undetectable, Early, Late, Responsive, Full Unresponsive, Partially Unresponsive, Data Benign, No Data, EDC, Non-EDC	0.893, 0.107, 0, 0, 0.891, 0.42, 0.67, 0.413, 0.109, 0.052, 0.426
Permanent	Instruction Replacement	In-Time, Undetectable, Early, Late, Responsive, Full Unresponsive, Partially Unresponsive, Data Benign, No Data, EDC, Non-EDC	0.274, 0.726, 0, 0, 0.378, 0.348, 0.274, 0, 0.726, 0, 0.274
	…		

3. 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:

- *Availability of reliability parameter keywords*. If a language to describe reliability concepts already exists, it is our aim to explore it before writing something new.
- *A template mechanism*. Single components tend to cluster into classes that share information. Defining templates of components could be helpful, potentially reducing the time required to describe a new component and guaranteeing that required information is properly described.
- *An 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 simplifying it. A clear drawback is that, resorting to templates, the readability by humans will be more complex.
- *Values as formula definition*. While most information can be count on precise values, we expect to describe some of them as formulas in order to define the final value as a function of other parameters.
- *Reliability related data metrics*. Common languages define very simple data types. As described in the previous sections, for some parameters we look for values to be associated to specific metrics.
- *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 within CLERECO. Our analysis also takes into account the possibility of re-using tools that have already been developed, thus reducing the effort to build tools within CLERECO. 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).

3.1. 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) [13]. 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 Internet as final platform. The design of XML focuses on documents but it is widely used for the representation of arbitrary data structures, [14].

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) [15]. 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 formula 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 tons of information can be found on the Internet.

3.2. UML

The Unified Modeling Language (UML) offers a way to visualize a 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, [16].

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 [17]. 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.

3.3. 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 [18][19], 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 specifies failure rate only, modeling of HW and SW components within the same system does not seems 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).

3.4. RIIF

The *Reliability Information Interchange Format* (RIIF) language is an application-specific language targeting the problem of modeling failure propagation in System on Chips (SoCs). RIIF was first proposed in [4] and was further developed during a dedicated workshop at Design and Test in Europe Conference 2013 (DATE'13) [22][23]. 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 [24]. 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][21][22][23]. The early stage of development of the RIIF language brings an opportunity to the CLERECO project, which most probably must be investigated.

3.5. Language 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 [21] 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 [21], 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.

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 **CERIIF (CLERECO Extended RIIF)** and it will be described in Section 4 of this deliverable.

4. CLERECO Extended RIIF language

This section presents the RIIF extensions introduced within CLERECO. With extensions here we both consider (a) new language keywords/statements or (b) specific usage of existing language statements in the context of the CLERECO project. In order to highlight the improvements introduced within CLERECO, we start with the description of the basic RIIF definitions to move later to the language improvements proposed in CLERECO.

In RIIF, the keyword *component* is useful to represent a system's HW component. From the CLERECO perspective, the same keyword could be re-used to define a SW component as well.

RIIF authors also defined two additional types of "entities" to represent and describe:

1. the **Requirements** to evaluate the components' model (*requirement* keyword), and
2. the **Environments** under which the whole model must be analyzed (*environment* keyword).

In practical terms, components, requirements and environments can be described starting from three simple declarations as:

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

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

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

where <LABEL> is a unique name to identify the instance of the entity.

To define the set of information characterizing a single component, RIIF offers two alternatives (hence keywords): a *constant* or a *parameter*. Since no full documentation is provided, the meaning of these two keywords can only be partially speculated. We consider that the purpose is to differentiate information to be used internally (*constant,* [4]) from information that must be exposed outside the component (*parameter,* [4]). The actual difference is the possibility to define aggregated information for a parameter, while the constant is expressed by a single value, only. 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 parameter. The following RIIF snapshot proposes an example of basic usage of constants and parameters applied to the definition of a register file:

```
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 share the **type** information to define the kind of value they store. The general way to define both of them is based on the following schema:

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

The type can be chosen among the following list: **boolean**, **integer**, **float**, **enum** (as for an enumerative of items), and **time** (to define timing related information). The value assignment 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).

Every time information from another component of system is required, it is possible to refer to it using a **getValue** function:

```
  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:

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

The *assign* keyword tells to 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*. Typically, an extra attribute comes with a failure mode: the rate.

```
      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* share 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*).

In the next subsections we describe a set of components, showing how we resort to RIIF to properly model them, and describing all improvements we proposed and implemented in the language.

4.1. CERIIF Templates

The first task to accomplish in order to reduce the effort required to describe a component is the definition of common information across HW and SW components or across classes of components belonging to the same group. The aim is to simplify the definition, reducing the amount of information to be re-written for each component, and to guarantee that essential information for a given component are properly provided.

In RIIF, the possibility to define templates (mainly to address components and sub-components) is delegated to the definition of a component in which common information items are described. In order to implement a full template mechanism (such as in most high level programming languages), CERIIF introduces a new *template* entity defined as follows:

```
template <LABEL>;
  …
endtemplate
```

Within a template it is possible to define a set of constants, parameters and failure modes that are common to all components implementing the template. Moreover resorting to the new CERIIF keyword (*abstract*) the actual values associated with constants and parameters do not necessarily need to be defined at this stage. Once a template is applied to a component, the user is required to define the actual values for the abstract items described within the template.

Referring to Section 2, a general CLERECO component (either HW or SW) can be defined according to the following template (the full version can be found in the Section 8.1 at the end of the document):

```
template CLERECO_COMPONENT;
    // Definition of common information
    abstract constant NAME: string;
    …
    abstract constant TYPE: enum {HW, SW};
    …
endtemplate
```

Each definition within a template is identified with the CERIIF keyword (*abstract*) and by formally defining the information type for a constant or a parameter.

Once a template is available, two possible usages are allowed in CERIIF:

1. The template can be further refined generating a more detailed and specific template through the CERIIF extension mechanism.
2. The template can be used to instantiate a component, thus implementing the template with final values.

Extending the template means defining a template inheriting all definitions contained in the parent one. The keyword for this mechanism is *extends*.

```
template CLERECO_HW_COMPONENT extends CLERECO_COMPONENT;
    // imposing a specific value for a constant (or a parameter)
    impose TYPE = HW;
    …
    abstract fail_mode ERRORE_RATE[];
endtemplate
```

In the example above, the new template (CLERECO_HW_COMPONENT) extends the previously defined CLERECO_COMPONENT. The new CERIFF keyword *impose* is designed to specify, at template level the value of a constant or of a parameter. Whenever a component needs to implement a template, the keyword *implements* is used:

```
component REGISTER_FILE implements CLERECO_HW_COMPONENT;
```

It means that all already defined information included in the template must be explicitly assigned to the component automatically.

When defining both templates and components it is often useful to define associative arrays of values. In [4], RIIF authors suggested the definition of simple vectors proposing a few examples in which vectors are defined as follow:

```
<keyword> <LABEL> [1..<MAX_NUMBER_OF_ITEMS>];
```

The maximum number of items in RIIF is known a priori and the items are referenced based on a numeric index. Since in many cases information is naturally describe as labels, with the ERROR_RATE (see Table 3) definition we suggest a new way to define vectors:

```
<keyword> <LABEL> [];
```

The empty brackets means two different things:

1. The maximum number is not known, thus elements can be freely "appended".

```
<LABEL>[<index_label>] = <value>;
```

2. The index of the item is defined during the element creation. In this way, there is no need to number the vector items and the access is based on the label used as an index.

```
<LABEL>[<index_label>]'<attribute> = <value>;
```

The next section shows examples of how Hardware and Software components are described using CERIIF. They implement one of the two templates that can be found in Section 8.1 describing an HW and a SW component. It is worth to mention here that at this stage of the project further refinements of the language are still possible to deal with specific requirements that will be identified during the development of the CLERECO EDA tool-suite. A more complete and precise definition of the language will be provided in the next release of this deliverable (D5.1.2) later in the project.

4.2. CERIIF Characterization of Components

Once defined two different templates, targeting main characteristics of HW and SW components, hereinafter we use them to describe three different components:

1. A generic register file.
2. A microprocessor (with and without protection mechanisms).
3. A software application (with and without software techniques for data protection).

All information described in Sections 2.1 and 2.2 are fit within the description. Full version of each description can be found in Section 8 as an Appendix of this deliverable. Figure 2 depicts how template and inheritance mechanisms will be used to efficiently describe all the considered components. The template mechanism allows the differentiation of HW and SW components (as already seen in the previous Section) and the inheritance mechanism helps the hierarchical organization of components (e.g., different version of the same component) and simplifies the component's descriptions.

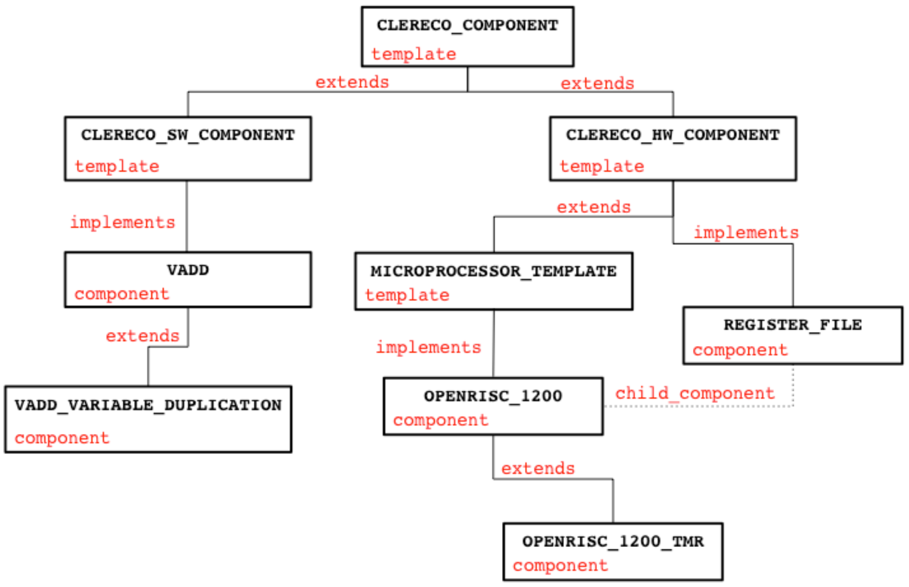

Figure 2 - Templates and Inheritance schema

Starting from the Register file, the main idea is implementing the CLERECO_HW_COMPONENT template. The first step is the definition of all constants and parameters defined in the template:

```
component REGISTER_FILE implements CLERECO_HW_COMPONENT;
    set NAME = "Generic Register File";
    …
    set NODE_SIZE = 0.18;
    assign NODE_SIZE'unit = "um";
    …
```

CERIIF extends RIIF by defining the *set* keyword used within a component to explicitly indicate when an *abstract* field is explicitly defined.

The number of registers and the number of flip-flops per register are two important information items that characterize a generic register file. They are therefore defined as parameters in the component:

```
    parameter NUMBER_REGISTERS: integer := 8;

    parameter FF_PER_REG: integer :=  32;

    constant SBU_TEMPERATURE_EFFECT_COEFF: float := 5.6e-12;

    // defining the current temperature...
    parameter CURR_TEMPERATURE: float;
    // ... the actual value will be assigned depending on the enviroment
defined.
    assign CURR_TEMPERATURE'value = environment.getValue(TEMPERATURE);

    assign  ERROR_RATES[SBU]'description = "Single bit upset" ;
    assign  ERROR_RATES[SBU]'unit = FITS;

    assign  ERROR_RATES[SBU]'rate =
(NUMBER_REGISTERS*FF_PER_REG/pow(2,20))*(SBU_TEMPERATURE_EFFECT_COEFF *
CURR_TEMPERATURE);
endcomponent
```

The constant SBU_TEMPERATURE_EFFECT_COEFF and the CURR_TEMPERATURE parameter are used in the above example to define the error rate for the Single Bit Upset (SBU) failure mode. It is interesting to note how the SBU failure mode is defined in the ERROR_RATES vector. The rate's formula is a simplified version (for reading purposes only) of the one found [27]. It is particularly important to underline the way the current temperature is evaluated: the *getValue* word indicates the intent to retrieve the actual value from the environment defined (as seen in previous section) by means of accessing its parameter TEMPERATURE.

The generic register file is a really simple component. In order to show a more complex definition we report in the next example the description of a generic microprocessor component. The modeling starts with the creation of template, including a parameter to describe the operation set (see Table 2) of a microprocessor, i.e., its Instruction Set Architecture (ISA).

```
template MICROPROCESSOR_TEMPLATE extends CLERECO_HW_COMPONENT;
    // definition of the Instruction Set Architecture through a table
type.
    abstract parameter ISA: table;
    // table comes with two attributes: its headers and the items
    impose ISA'headers = { NAME, TYPE, TIMING, INVOLVED_FF, SBF_P_MASK,
STUCKAT_P_MASK };
endtemplate
```

The template extends the hardware component template, including the ISA parameter of *table* type. The table type is a further improvement within CERIIF. The idea is to be able to describe structured data organized as tables of information. A table is defined by two attributes: the *header* and the *items*. Headers include a vector (instantiated inline using comma separated items within {} brackets) of all columns index labels. The number of elements of the vector sets the headers' dimension dynamically. Since each microprocessor owns a different set of instructions, the items definition is demanded to the component implementation.

In this specific example we propose a snippet of code describing the OpenRISC1200 [25], where the first part of the implementation includes the *set* of all constants and parameters already defined in the templates:

```
component OPENRISC_1200 implements MICROPROCESSOR_TEMPLATE;
    set NAME = "OpenRISC 1200";
    …
```

Then the custom definition of the component starts by defining few constants regarding raw error rates for failure modes and including a subcomponent: the register file.

```
    constant RAW_SBU: float := 1.2e-20;
    constant RAW_STUCK_AT: float := 1.5e-25;
    child_component REGISTER_FILE GPR;
    assign GPR.SIZE = 32;
```

The *child_component* keyword is already present in the original version of RIIF. It helps aggregating sub-components, creating more complex structures. Once instantiated a sub-component, it is possible to access its parameters (not its constants) to customize the instance, e.g., the size of the register file.

The sub-component will be useful to define the number of flip-flops involved in an instruction execution (INVOLVED_FF field of each item of the ISA table). Thus, in the definition of an item, we resort to the number of flip-flops per register (FF_PER_REG parameter of the generic register file component):

```
    assign ISA'items = {
        [ "ADD", "add", 1, 3*GPR.FF_PER_REG, 0.001, 0.001],
        [ "BNE", "branch, 1, GPR.FF_PER_REG, 0.015, 0.015],
        [ "MULTU", "mul op", 10, 4*GPR.FF_PER_REG, 0.25, 0.10],
            [ "SLL", "shift", 1, 2*GPR.FF_PER_REG, 0.001, 0.001]
     };
```

The access to a single item of a table can be shown resorting to the definition of one failure mode:

```
     assign ERROR_RATE[SBU]'type = "transient",
     assign ERROR_RATE[SBU]'description = "Single Bit Upset";
     assign ERROR_RATE[SBU]'unit = FITS;
     assign ERROR_RATE[SBU]'affected = ISA;
     assign ERROR_RATE[SBU]'rate = (P_SBU *
(ISA[#][INVOLVED_REGISTERS_AREA] / AREA));
     assign ERROR_RATE[SBU]'timing_effect = ISA[#][TIMING] +
ISA[#][TIMING] * 0.25;
```

The SBU error rate is described resorting to a set of specific attributes:

- *affected* defines what set of operations is affected by the failure mode.
- *rate* defines the final rate of the failure mode (as in RIIF original version).
- *timing_effect* defines how the failure mode modifies the timing of operations (if applicable).

To simplify the formula statement in presence of a table, CERIIF introduces the [#] operator, which means "for each element in the table". As example, referring to the timing effect previously defined, the formula expresses that the timing of each item of the ISA is changed (in presence of a SBU failure) by a delay of 25% of its standard timing.

The inheritance mechanism enables to easily generate new complex components as extension of previously defined ones. As an example, we model a new version of the OpenRISC 1200 to include a mitigation mechanism, as described in Table 4:

```
component OPENRISC_1200_TMR extends OPENRISC_1200;
     …
     // Create a constant to define the penalty required to execute a TMR
mechanism
     constant TMR_PENALITY: integer := 5;

     // assign to each TIMING column value, the OPENRISC_1200 original
value incremented by the time required to perform TMR;
     // self is a new keyword...
     assign ISA'items[#][TIMING] = self + TMR_PENALITY;

     // reset the error rate of all failure mode because the TMR solve
them.
     assign ERROR_RATE[SBU]'rate = 0;
     …
endcomponent
```

For demonstration purposes only, we simplify the way a Triple Redundancy Mechanism (TMR) affect the microprocessor model. The idea is inheriting all previous definitions and modifying only the aspects that change in the new version. Then, looking at the description, we introduce the *self* keyword to help updating some already defined value. In the example, all TIMING are incremented of a delay value due to the extra time required to compute the operation three times and vote among them. Since the TMR technique aims at avoiding errors, the SBU rate can be set to 0. Through the use of inheritance the main advantage is that the description of a child component becomes very small therefore reducing the probability of errors.

Eventually, we describe a software component: an application program able to add two arrays consisting of 10,000 integer elements and monitoring the sum at the end of the calculations. The program is compiled to run under an LLVM based simulator, as described in Deliverable D4.1 (*Software Impact on system reliability: metrics and models*).

According to the software component template, the very first part of the description implements all information already defined, including the size and the type of implemented data protection mechanisms (see Section 8.1 for the template definition):

```
component VADD implements CLERECO_SW_COMPONENT;
    set NAME = "Vector ADD";
    set VENDOR = "CRNS";

    …
    set SIZE = 534;

    set PROTECTION = NONE;
    …
```

Thus we are able to describe a large number of the parameters identified in Table 6. It can be noticed how simple is the parameterization of such characteristics (in this example, through the number of items inside the vector):

```
    constant NUMBER_OF_ITEMS: integer := 10000;

    assign READING_ACCESSES = 76 * NUMBER_OF_ITEMS/10000;
```

Also dealing with the timing constraints of Table 7 is quite simple. In the template, we defined the table and its headers:

```
    abstract parameter TIMING_CONSTRAINS: table;
    impose TIMING_CONSTRAINS'headers = { WORKLOAD, EXEC_TIME, MAX_TIME,
AVG_TIME };
```

Then in the component only items must be defined:

```
    assign TIMING_CONSTRAINS'items = {
                    [ "TEST_BENCH1", 0.0000001, 2, 0.000001 ],
                    };
```

Similarly, the same description approach is applied to the Software Faulty Behaviors (SFBs) of Table 8. The parameters definition is contained in the template:

```
    abstract parameter SFB: table;
    impose SFB'headers = { SFM, OCCURRING_SFB, OCCURRING_SFB_P };
```

and the actual *items* are defied within the component:

```
    assign SFB'items = {
          [ "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} ],
          [ "permanent", "INSTR_REPLACEMENT", SFB_ITEMS, {0.274, 0.726,
0, 0, 0.378, 0.348, 0.274, 0, 0.726, 0, 0.274}],
          [ "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} ],
          [ "transient", "INSTR_REPLACEMENT ", SFB_ITEMS, {0.614, 0.309,
0, 0.077, 0.309, 0, 0.691, 0.691, 0.309, 0, 0}],
                    };
```

We resort to the CERIIF language flexibility to ease the description of the SBFs occurrences in presence of software fault models: the OCCURRING_SFB and OCCURRING_SFB_P fields of each row contain a vector. This way we implement the ability to list all SFBs that may occur (the SFB_ITEMS is a vector already defined in the template) and the list of all associated probabilities.

Similarly to the hardware component, we show how the inheritance mechanism is useful to easily extend a component with some new features. We implement the same algorithm with the ability of activating two different data protection mechanisms: variable duplication and variable triplication.

Basically, since the data protection mechanisms allow almost the same protection, we resort to the ability of RIIF language of declaring a value referred to a *if* clause, to distinguish the impact of each mechanism on the timing constraints:

```
 assign TIMING_CONSTRAINS'items = {
                 (PROTECTION == VAR_TRIP)?[ "TEST_BENCH1", self + 0.001,
self + 0.2 , self + 0.0015] : [ "TEST_BENCH1", self + 0.0021, self +
0.28, self + 0.0018 ]
                                };
```

While some of the masking probabilities and the software faulty behavior probabilities are going to be affected by the data protection mechanisms, we override only the items that show differences:

```
assign SFB'items = {
          [ "permanent", "WRONG_DATA", SFB_ITEMS, {0.88, 0.067, 0, 0.53,
0.044, 0.029, 0.927, 0.737, 0.073, 0.025, 0.165} ],
          [ "permanent", "INSTR_REPLACEMENT", SFB_ITEMS, {0.266, 0.726,
0, 0.008, 0.446, 0.28, 0.274, 0, 0.726, 0, 0.274}],
          [ "transient", "WRONG_DATA", SFB_ITEMS, {0.907, 0.009, 0,
0.084, 0.003, 0.001, 0.996, 0.986, 0.004, 0, 0.010} ],
          [ "transient", "INSTR_REPLACEMENT", SFB_ITEMS, {0.518, 0.309,
0, 0.173, 0.309,  0, 0.691, 0.691, 0.309, 0, 0}],
                          };
```

Due to inheritance mechanisms, we expect to maintain all items that are defined in the parent component.

5. Conclusion

This deliverables represents the first steps performed in CLERECO toward the aim of providing a formal and coherent description of all reliability aspects of a system. In particular, this deliverable focuses on the description of single components of a system. This deliverable provided two major contributions to the project. First the definition of an initial taxonomy of parameters required to describe reliability aspects of both hardware and software components. Second it introduces a new description language named CERIIF based on a previous existing language (RIIF) that enables a formal representation of these parameters. The CERIIF language will be used for tools developed within CLERECO to properly share and transfer reliability related information. The definition of CERIIF is still not final and modifications will be proposed when specific problems will be addressed. A final definition of this language and related tools (e.g., a full CERIIF parser) will be available in deliverable D.5.1.2.

6. Additional material on CLERECO SVN Repository

This section provides a link to tools, code and models developed in the framework of the activities described in this deliverable that are available through the CLERECO SVN Repository. This material, listed in Table 11 must be considered as integral part of the deliverable.

The CLERECO SVN repository is accessible through a web browse clicking on the links reported in Table 11. The access to the material requires authentication. Reviewers can access it using the following credentials:

- **Username**: clerecoreviewers
- **Password**: fp7-611404

Table 11: Additional Material

Item No.	Description	Link to the CLERECO SVN Repository
AM1	Full list of CERIIF files produced while characterizing HW and SW components	http://goo.gl/dRaZ0m

7. Bibliography

[1] 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.

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

[3] 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.

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

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

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

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

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

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

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

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

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

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

[20] 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

[21] 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

[22] 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.

[23] 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.

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

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

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

[27] 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.

8. Appendix

8.1. The CLERECO RIIF Templates

```
template CLERECO_COMPONENT;
    // Definition of common information
    abstract constant NAME: string;
    abstract constant VENDOR: string;
    abstract constant TYPE: enum {HW, SW};
    abstract constant CLASS: string;
    abstract constant SUBCLASS: string;
endtemplate
```

```
template CLERECO_HW_COMPONENT extends CLERECO_COMPONENT;
    // imposing a specific value for a constant (or a parameter)
    impose TYPE = HW;
    abstract constant TECHNOLOGY: enum { CMOS, FINFET };
    abstract parameter AREA: float;
    abstract parameter NODE_SIZE: float;

    abstract fail_mode ERRORE_RATE[];
endtemplate
```

```
template CLERECO_SW_COMPONENT extends CLERECO_COMPONENT
    // imposing a specific value for a constant (or a parameter)
    impose TYPE = SW;
    // ------------------------ Constant Declaration ------------------
-----
    abstract constant SIZE: integer;
    abstract constant PROTECTION: enum { NONE, VAR_DUP, VAR_TRIP};
    abstract constant SFB_ITEMS[1..11] := { IN_TIME, UNDETECTABLE,
EARLY, LATE, FULL_UNRESPONSIVE, PARTIAL_UNRESPONSIVE, RESPONSIVE,
DATA_BENIGN, NO_DATA, EDC, NON-EDC};

    abstract parameter READING_ACCESSES: integer;
    abstract parameter WRITING_ACCESSES: integer;
    abstract parameter MEMORY_ACCESSES: integer;
    abstract parameter NUMBER_OF_LOOPS: integer;
    abstract parameter ALGORITHM_COMPLEXITY: string;

    abstract parameter TIMING_CONSTRAINS: table;
    impose TIMING_CONSTRAINS'headers = { WORKLOAD, EXEC_TIME, MAX_TIME,
AVG_TIME };

    abstract parameter SFB: table;
    impose SFB'headers = { SFM_TYPE, SFM, OCCURRING_SFB, OCCURRING_SFB_P
};
endtemplate
```

```
template MICROPROCESSOR_TEMPLATE extends CLERECO_HW_COMPONENT;
    // definition of the Instruction Set Architecture through a table
type.
    abstract parameter ISA: table;
    // table comes with two attributes: its headers and the items
    impose ISA'headers = { NAME, TYPE, TIMING, INVOLVED_FF, SBF_P_MASK,
STUCKAT_P_MASK };
endtemplate
```

8.2. CLERECO Hardware Component Examples

```
component REGISTER_FILE implements CLERECO_HW_COMPONENT;
    set NAME = "Generic Register File";
    set VENDOR = "Generic Vendor";

    set CLASS = "Memory";
    set SUBCLASS = "Registers File";

    set TECHNOLOGY = CMOS;
    set NODE_SIZE = 0.18;
    assign NODE_SIZE'unit = um;

    set AREA = 0.03;
    assign  AREA'unit = mm2;

    parameter NUMBER_REGISTERS: integer := 8;

    parameter FF_PER_REG: integer :=  32;

    constant SBU_TEMPERATURE_EFFECT_COEFF: float := 5.6e-12;

    // defining the current temperature...
    parameter CURR_TEMPERATURE: float;
    // ... the actual value will be assigned depending on the enviroment
defined.
    assign CURR_TEMPERATURE'value = environment.getValue(TEMPERATURE);

    assign  ERROR_RATES[SBU]'description = "Single bit upset" ;
    assign  ERROR_RATES[SBU]'unit = FITS;

    assign  ERROR_RATES[SBU]'rate =
(NUMBER_REGISTERS*FF_PER_REG/pow(2,20))*(SBU_TEMPERATURE_EFFECT_COEFF *
CURR_TEMPERATURE);
endcomponent
```

```
component OPENRISC_1200 implements MICROPROCESSOR_TEMPLATE;
    // set all inherited definitions of constants & parameters
    set NAME = "OpenRISC 1200";
    set VENDOR = "Opencores.Org";

    set CLASS = "Microprocessor";
    set SUBCLASS = "RISC";

    set TECHNOLOGY = CMOS;
    set NODE_SIZE = 0.18;
    assign NODE_SIZE'unit = um;
    set AREA = 0.5;
    assign AREA'unit = mm2;

    // Define RAW Error Rate for Single Bit Upset (SBU)
    constant RAW_SBU: float := 1.2e-20;
    // Define RAW Error Rate for Stuck-at fault
    constant RAW_STUCK_AT: float := 1.5e-25;

    child_component REGISTER_FILE GPR;
    assign GPR.SIZE = 32;

    // assign a description to the ISA table
    assign ISA'description = "The instruction Set Table";

    // assign the items in the form of vector of associative vectors
(index label defined in the headers attributes)
    assign ISA'items = {
          [ "ADD", "add",   1, 3*GPR.FF_PER_REG, 0.001, 0.001],
          [ "BNE", "branch, 1, GPR.FF_PER_REG, 0.015, 0.015],
          [ "MULTU", "mul op", 10, 4*GPR.FF_PER_REG,      0.25, 0.10],
          [ "SLL", "shift", 1, 2*GPR.FF_PER_REG, 0.001, 0.001]
    };

    // -------------------------------------- Define Failure Modes -----
----------------------------
    assign ERROR_RATE[SBU]'type = "transient",
    assign ERROR_RATE[SBU]'description = "Single Bit Upset";
    assign ERROR_RATE[SBU]'unit = FITS;
    assign ERROR_RATE[SBU]'affected = ISA;
    assign ERROR_RATE[SBU]'rate = (P_SBU *
(ISA[#][INVOLVED_REGISTERS_AREA] / AREA));
    assign ERROR_RATE[SBU]'timing_effect = ISA[#][TIMING] +
ISA[#][TIMING] * 0.25;

    assign ERROR_RATE[STUCK_AT]'type = "permanent";
    assign ERROR_RATE[STUCK_AT]'description = "stuck-at in a single bit
of a register";
    assign ERROR_RATE[STUCK_AT]'unit = FITS;
    // to define 'affected...
    assign ERROR_RATE[STUCK_AT]'affected = ISA;
    assign ERROR_RATE[STUCK_AT]'rate = RAW_STUCK_AT * (1-
ISA[#][STUCKAT_P_MASK]);
    // to define timing_effect
    assign ERROR_RATE[STUCK_AT]'timing_effect = 0;
endcomponent
```

```
component OPENRISC_1200_TMR extends OPENRISC_1200;
    // Change the area according to the extra-space needed to implement
TMR
    set AREA = 0.6;

    // Create a constant to define the penalty required to execute a TMR
mechanism
    constant TMR_PENALITY: integer := 5;
    // assign to each TIMING column value, the OPENRISC_1200 original
value incremented by the time required to perform TMR;
    // self is a new keyword...
    assign ISA'items[#][TIMING] = self + TMR_PENALITY;

    // reset the error rate of all failure mode because the TMR solve
them.
    assign ERROR_RATE[STUCK_AT]'rate = 0;
    assign ERROR_RATE[SBU]'rate = 0;
endcomponent
```

8.3. CLERECO Software Component Examples

```
component VADD implements CLERECO_SW_COMPONENT;
    set NAME = "Vector ADD";
    set VENDOR = "CRNS";

    set CLASS = "Application";
    set SUBCLASS = "Encoding Algorithm";
    set SIZE = 534;

    set PROTECTION = NONE;

    constant NUMBER_OF_ITEMS: integer := 10000;

    assign READING_ACCESSES = 76 * NUMBER_OF_ITEMS/10000;
    assign WRITING_ACCESSES = 75 * NUMBER_OF_ITEMS/10000;
    assign MEMORY_ACCESSES = 151 * NUMBER_OF_ITEMS/10000;
    assign NUMBER_OF_LOOPS = 3;
    assign ALGORITHM_COMPLEXITY = "n";

    assign TIMING_CONSTRAINS'items = {
                     [ "TEST_BENCH1", 0.0000001, 2, 0.000001 ]
                     };

    assign SFB'items = {
          [ "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} ],
          [ "permanent", "INSTR_REPLACEMENT", SFB_ITEMS, {0.274, 0.726,
0, 0, 0.378, 0.348, 0.274, 0, 0.726, 0, 0.274}],
          [ "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} ],
          [ "transient", "INSTR_REPLACEMENT ", SFB_ITEMS, {0.614, 0.309,
0, 0.077, 0.309, 0, 0.691, 0.691, 0.309, 0, 0}]
                     };
endcomponent
```

```
component VADD_VARIABLE_DUPLICATION extends VADD;
    set NAME = "Vector ADD with Variable Duplication";

    set PROTECTION = VAR_DUP;

    assign TIMING_CONSTRAINS'items = {
```

```
                (PROTECTION == VAR_TRIP)?[ "TEST_BENCH1", self + 0.001,
self + 0.2 , self + 0.0015] : [ "TEST_BENCH1", self + 0.0021, self +
0.28, self + 0.0018 ]
                              };

    assign SFB'items = {
          [ "permanent", "WRONG_DATA", SFB_ITEMS, {0.88, 0.067, 0, 0.53,
0.044, 0.029, 0.927, 0.737, 0.073, 0.025, 0.165} ],
          [ "permanent", "INSTR_REPLACEMENT", SFB_ITEMS, {0.266, 0.726,
0, 0.008, 0.446, 0.28, 0.274, 0, 0.726, 0, 0.274}],
          [ "transient", "WRONG_DATA", SFB_ITEMS, {0.907, 0.009, 0,
0.084, 0.003, 0.001, 0.996, 0.986, 0.004, 0, 0.010} ],
          [ "transient", "INSTR_REPLACEMENT", SFB_ITEMS, {0.518, 0.309,
0, 0.173, 0.309,  0, 0.691, 0.691, 0.309, 0, 0}]
                        };
endcomponent
```