RIIF-2: toward the next generation Reliability Information Interchange Format

A. Savino, S. Di Carlo, A. Vallero, G. Politano
Dep. of Control and Computer Engineering
Politecnico di Torino
Torino, Italy
e-mail: <firstname>.<lastname>@polito.it

D. Gizopoulos
Department of Informatics and Telecommunications
University of Athens
Athens, Greece
e-mail: dgizop@di.uoa.gr

A. Evans
IROC Technologies
Grenoble, France
e-mail: adrian.evans@iroctech.com

Abstract[1]— **This paper describes the joint effort of the two FP7 EU projects CLERECO and MoRV toward the definition of an extended reliability information exchange format able to manage reliability information for the full system stack, from technology up to the software level. The paper starts from the RIIF language initiative, proposing a set of new features to improve the expression power of the language and to extend it to the software layer of a system. The proposed extended reliability information exchange format named RIIF-2 has the potential to support the development of next generation reliability analysis tools that will help to fully include reliability evaluation into an automated design flow, pushing cross-layer reliability considerations at the same level of importance as area, timing and power consumption when performing design exploration for new products.**

Keywords— reliability, robustness, design flow, modeling languages

I. Introduction

Reliability has always been a serious issue for IC designers, recently exacerbated by the innovations required to continue transistor miniaturization such as FinFET transistors, high-k gate dielectrics, etc. [1]. At current levels of integration, reliability is a fundamental design dimension that must be considered early in the design flow together with area, timing and power. Failing to meet necessary reliability requirements may add excessive re-design costs to recover and may generate severe consequences on the success and profitability of a product [2]. Today, reliability analysis is mostly focused at the technology and circuit level. Excessive margining and overdesign are often required to show a product will operate safely over its full lifetime. However, technology scaling is evolving so quickly that this approach is impacting the performance, area, and power benefits of new technologies to a level that makes reliability oriented overdesign economically not sustainable [3]. To overcome this limit, cross-layer approaches in which reliability levels of complex systems are sustained by techniques that operate at all levels of the system stack (i.e., technology, hardware architecture and software) have been proposed [4][5][6][7][8]. The research community often focuses on efficient implementations of cross-layer approaches for a specific sample design. However, tools for reliability analysis are still at their early stages compared to other very mature EDA design tools and this represents a major issue for mainstream applications. This lack of tools is partly the result of a lack of standardization of file formats and the lack of powerful languages to express reliability information at all levels of the system stack [9].

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 optimization 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. An overly simplified reliability description formalism often used for system level reliability analysis is Reliability Block Diagrams (RBDs) [10][11]. 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. More recently we have observed an increased call for the definition of standardized languages allowing the formalization, specification and modeling of extra-functional reliability properties to enable a major productivity improvement in the design of fault tolerant systems by integrating automatic reliability analysis within the existing silicon design flow. Among the different projects, the RIIF (Reliability Information Interchange Format) initiative launched by iROC Technologies and supported by the FP7 EU Project MoRV (Modeling Reliability under Variability) is gaining a major interest from the research community [12][9]. RIIF is a machine-readable format able to describe the failure mechanisms associated with a generic hardware component as well as a basic hardware system, through the decomposition of complex components into simpler ones. In particular, using RIIF, failure modes of a component can be expressed as

[1] This paper has been fully supported by the 7th Framework Program of the European Union through the CLERECO Project (http://www.clereco.eu), under Grant Agreement 611404 and through the MoRV Project (http://morv-project.eu), under Grant Agreement 619234.

functions of its related parameters. Despite taking a significant step toward the definition of a reliability management description language, RIIF still has some limitations and room for improvement. In particular, the initial RIIF model mainly focused on the description of hardware related reliability information without considering the higher layers of the stack, which may require further extensions of the language in order to cope with their complexity.

In this context, the FP7 EU project CLERECO (Cross-Layer Early Reliability Evaluation for the Computing cOntinuum), which focuses on the development of tools and methods for early reliability analysis of complex digital systems, has joined forces with the MoRV project working on an revised version of the RIIF language (RIIF-2) able to model and manage reliability information with a wider scope. In particular the introduced extensions take into account that a system is composed of both hardware and software components, and the language must be broad enough to encapsulate the effect of both categories of components and to link information among layers in order to properly describe how errors propagate and are handled within the system. This way, it is possible to distribute the reliability analysis throughout the design flow of a system and to propagate information across different levels of the system stack. Moreover, when introducing all levels of the system stack the volume of information increases significantly. The introduction of concepts such as inheritance and advanced templates, common in high-level programming languages, will help in effectively managing the information and overcoming the limitations of the simple template mechanism available in the original RIIF definition. This paper summarizes the result of this joint effort providing a short overview of the basic RIIF concepts as originally developed and focusing on the proposed extensions to move the language toward the management of reliability information for the full system stack.

The paper is organized as follows: section II reviews the basic RIIF concepts and describes the main extensions being proposed, whereas section III focuses on how the extended RIIF language can be used to model reliability information for software components. Finally section IV provides information on the status of the implementation of the extended language and section V summarizes the main contributions and concludes the paper.

II. RIIF LANGUAGE EXTENSION

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. This section focuses on the first of these two goals. 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 evaluation. 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.

A. Brief RIIF Overview

This section reviews the basic RIIF concepts. Interested readers may refer to [12] 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 has to meet (**`requirement`** keyword), and (2) the environment under which the component is going to operate (**`environment`** keyword) whose characteristics can be used to express different parameters of a components (e.g., the error rate of a device is a function of the temperature whose range is defined based on the environment). 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. 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. Fig. 1 outlines an example of the basic usage of constants and parameters applied to the definition of a simple SRAM component. Both parameters and constants are defined by a label (e.g., `VOLTAGE` in Fig. 1) and by a data type defining the kind of information they store. Examples of allowed data types are: **`boolean`**, **`integer`**, **`float`**, **`enum`** (for enumerative items), **`time`** (to define timing related information), etc. The value assignment is optional (it can be set later in the definition) and the actual value can be either explicit or a formula (`SBU'RATE` at line 27 of Fig. 1 is expressed in terms of other constants and parameters).

The **`assign`** keyword is used to assign a value to a parameter or to an attribute of a parameter (referred through a tick(') and its label). Attributes of a parameter are free, and enable to specify simple aggregated information, such as units (metrics) and descriptions. Users can define failure modes in a similar fashion as parameters. As an example, Fig. 1 shows the definition of three failure modes (`SBU`, `MBU` and `SEFI`) using the **`fail_mode`** keyword that helps to distinguish between general parameters and failure modes.

```
component SIMPLE_SRAM;

   // Parameter Declaration
   parameter VOLTAGE : float := 1.0;
   assign VOLTAGE'UNITS = VOLTS;
   parameter DIE_TEMP : float := 25.0;
   assign DIE_TEMP'UNITS = CELSIUS;

   // Parameter to be modified by user
   parameter NUM_BITS : integer := 1024*1024; //number of bits

   // Constants specific to modeling this SRAM
   constant A_DIFF : float    := 3.2;
   constant Q_COL_EFF : float := 0.6;
   constant MBU_RATIO : float := 0.25;

   // Define Radiation Induced Failure Modes
   fail_mode SBU;  // Rad. induced single bit error
   fail_mode MBU;  // Rad. induced multiple bit error (same word)
   fail_mode SEFI; // Radiation induced failure of entire device

   assign SBU'UNITS = FITS;
   assign MBU'UNITS = FITS;
   assign SEFI'UNITS = FITS;

   // Equations to specify rate of defined failure modes
   assign SBU'RATE = NUM_BITS * A_DIFF * EXP( - CORE_VOLTAGE /
                     Q_COL_EFF );
   assign MBU'RATE = SBE'RATE * MBU_RATIO;
   assign SEFI'RATE = 10; // obtained from testing

endcomponent SIMPLE_SRAM
```

Fig. 1. RIIF description of a simple SRAM component.

B. RIIF-2 Extensions

In RIIF, the possibility to define templates is limited to the definition of a hardware component in which common desired information is listed without providing 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 simplify 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 to define a set of constants, parameters and failure modes that must be defined in all components implementing the template. The example of Fig. 2 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 define voltage, temperature and size
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 = CELSIUS;
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
```

Fig. 2. 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 Fig. 2 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 Fig. 4.

Further language extensions proposed in this paper focus on the introduction of *complex data structures*. The RIIF language only supports the definition of fixed size numerically indexed arrays. However, in several cases information must be associated to a set of labels to make it easy its 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 Fig. 2. 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 Fig. 3 and Fig. 4 later in the paper. 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.

III. Software components characterization

The software layer (i.e., firmware, system and application software) is a key element of a complex digital system and plays a key role from the reliability standpoint [14]. While raw errors are generated at the technology level due to undesired effects such as ageing, environmental stress, variability etc., many detectable errors can be gracefully managed by correct software handling and a significant portion of the undetectable errors may be masked, depending on the application. The software routines executed in a complex system play a significant role in introducing this type of masking effects and must be properly characterized when implementing cross-layer reliability solutions to potentially reduce the design margins imposed at the hardware level. 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.

This section overviews how the expression power of RIIF-2 can be used to model and manage reliability related information for software components, thus addressing the full system stack. In order to properly model software components in RIIF-2 we started from the identification of a preliminary set of general parameters required to express basic properties of a software component that may influence its ability to mask or propagate hardware faults:

- *software class* to distinguish among firmware, system and application software, etc. Each class can be further split into *subclasses*.
- *code size*. It can be expressed as: (i) the number of high level code lines, (ii) the number of assembly instructions, (iii) the size of the binary code. Moreover, it can be statically computed or dynamically computed based on the execution of a set of workloads.
- *number of loops* to count how many loops are statically defined in the software code and how many times they are executed for selected workloads.
- *variable lifetime* to identify those variables with longer life and therefore higher chance to be corrupted. It could be an average value or a distribution function.
- *algorithm complexity* computed in general as a function of other parameters such as size, number of loops, etc.
- *read access rate/count* to count the memory read operations. It can be statically or dynamically computed.
- *write access rate/count* to count the memory write operations similarly to the read access rate.
- *cache miss rate/count* to count cache misses. It can be collected through dynamic analysis resorting to hardware facilities available in most commercial microprocessors.
- *cache hit rate/count* to count the number of cache hits.
- *accessed memory pages* to count the number of memory pages accessed during the execution of the code.

Together with these general parameters, a set of more specific parameters can be collected and modeled. Hardware failures may significantly affect timing of a software application. Therefore, it is important to be able to model *timing constraints* that must be respected to obtain a correct result. Since the software execution time depends on the workload, timing constraints are usually defined as an *expected execution time* in relation to a given *workload*. Moreover, if margins can be accepted in the execution time, optional *max accepted execution time* and *average accepted execution time* properties can be defined. Hardware level failure modes (e.g., SBU) may deviate the correct software execution generating a set of possible *software faulty behaviors* (SFBs). In order to model the way a hardware failure enters the software domain and is propagated eventually generating a SFB we resort to the concept of *software fault models* (SFMs). We consider the SFMs has introduced by Vallero et. al in [7] that translate 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. We consider three main classes of SFBs: timing (e.g., early/late execution), data (e.g., incorrect data, no data produced, etc.), and responsiveness (e.g., responsive, partial responsive, fully unresponsive, etc.).

Characterizing the SFBs of a software component means providing the probability of observing a behavior in the presence of one or a combination of SFMs. Due to the limited space available in this paper, the reader may refer to [7] for a detailed description of the concept of SFM and for a preliminary taxonomy.

Fig. 3 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 template formalism introduced in section II.B. 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 [13]. 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 Fig. 3. 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.

Fig. 4 (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, Fig. 4 (lines 13-20) shows another way to describe the component proposed in Fig. 4. In this case we use the iterative operator [#] introduced in section II.B 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.

IV. Implementation

The definition of the RIIF-2 language is still underway. This prevent the availability of a complete toolchain to support the use of the language in both research and commercial tools. To promote the RIIF-2 initiative and to stimulate the contribution of the reliability community a public website available at http://riif.iroctech.com/ has been setup. This will serve as a central repository for the initiative where news and information will be published, tools will be released. Researchers interested in collaborating on the initiative can also refer to this website to join the RIIF-2 development team.

V. Conclusions

In this paper we presented the effort of two FP7 European projects (MoRV and CLERECO) toward the definition of a standardized language for modeling reliability information for complex digital systems taking into account the full system stack. Starting from the RIIF initiative, this paper introduces an extended language that includes a set of features able to increase the expressive power of the language with particular emphasis on its use in automatic reliability analysis tools. Moreover, the paper focuses on exploiting the new extended language to cover the characterization of reliability information for software modules that are of primary importance when cross-layer mechanisms must be analyzed and implemented for a new system. Through the joint dissemination end exploitation effort of two FP7 EU projects the extended RIIF language will support the development of next generation reliability analysis tools that will help to include reliability evaluation into an automated design flow at the same level as area, timing and power consumption.

```
component VECTORCALC_CORE;
  // An hardware component performing vectorial calculations
 ...
  parameter fail_mode SBU;
  assign SBU'RATE = 10; //obtained from radiation tests
 ...
  parameter SW_FM: table;
  assign SW_FM'HEADERS = {FAILMODE, SFM, RATE};
  assign SW_FM'ITEMS = { // Obtained from fault injection
       [ "SBU", "WRONG_DATA", 0.3 * SBU'RATE ],
       [ "SBU", "WRONG_INSTRUCTION", 0.2 * SBU'RATE ], ... };
endcomponent

template SW_COMPONENT;
  // All programs must define the name, size, …
  abstract parameter         NAME : string;
  abstract parameter         SIZE : integer;
  abstract parameter        LOOPS : integer ;
  abstract parameter   PROTECTION : enum {NONE, VAR_DUP, ...};
  abstract parameter  READ_ACCESS : integer ;
  abstract parameter WRITE_ACCESS : integer ;
  abstract platform   executed_on;

  // List of possible SFB considered in our library
  abstract constant    SFB_LIST:= {IN_TIME, UNDETECTABLE, EARLY,
  LATE, FULL_UNRESPONSIVE, PARTIAL_UNRESPONSIVE, RESPONSIVE,
  DATA_BENIGN, NO_DATA, EDC, NON-EDC};

  // Timing constraints depending on the workload
  abstract parameter TIMING_CONSTRAINTS   : table;
  impose             TIMING_CONSTRAINTS'HEADERS = {WORKLOAD,
                     EXEC_TIM, MAX_TIME, AVG_TIME};

  // Software faulty behaviors table defining the probability
  of occurrence of each SFB given the occurrence of each SFM.
  abstract parameter SFB                : table;
  impose             SFB'HEADERS = {SFM_TYPE, SFM,
                     OCCURING_SFB, OCCURRING_SFB_RATE};
endtemplate

component VADD implements SW_COMPONENT;
  set NAME                   = "Vector ADD";
  set SIZE                   = 524;
  set SIZE'UNITS             = instructions;
  set PROTECTION             = NONE;
  constant NUMBER_OF_ITEMS := 10000;
  set READ_ACCESS            = 76 * NUMBER_OF_ITEMS / 10000;
  set WRITE_ACCESS           = 75 * NUMBER_OF_ITEMS / 10000;
  set LOOPS                  = 3;
  set executed_on            = VECTORCALC_CORE;
  child_component            VPRINT;
  .....
  assign TIMING_CONSTRAINTS'ITEMS = {
    [ TEST_BENCH1, 0.0000001, 2, 0.000001 ],
    [ TEST_BENCH2, 0.0000003, 2.1, 0.0000004 ] };
  assign SFB'ITEMS = {
    [ "permanent", "WRONG_DATA", SFB_ITEMS, { 0.893, 0.107, 0,
    0, 0.891, 0.042, 0.067, 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
```

Fig. 3. System modeling with both hardware and software components.

```
component VADD_VARIABLE_DUPLICATION_VER1 extends VADD;
  set NAME = "Vector ADD with Variable Duplication";
  set PROTECTION = VAR_DUP;
  assign TIMING_CONSTRAINS'ITEMS = {
   (PROTECTION == VAR_DUP)?
   [ "TEST_BENCH1", self+0.001, self+0.2, self+0.0015] :
   [ "TEST_BENCH1", self , self , self ],
   (PROTECTION == VAR_DUP)?
   [ "TEST_BENCH2", self+0.001, self+0.2, self+0.0015] :
   [ "TEST_BENCH2", self , self , self ]};
endcomponent

component VADD_VARIABLE_DUPLICATION_VER2 extends VADD;
 ...
  assign TIMING_CONSTRAINS'ITEMS[#][EXEC_TIME]=
   (PROTECTION == VAR_DUP)? self + 0.001 : self ;
  assign TIMING_CONSTRAINS'ITEMS [#][MAX_TIME]=
   (PROTECTION == VAR_DUP)? self + 0.2 : self ;
  assign TIMING_CONSTRAINS'ITEMS[#][AVG_TIME]=
   (PROTECTION == VAR_DUP)? self + 0.0015 : self ; };
 ...
endcomponent
```

Fig. 4. Vector_ADD software with protection mechanisms.

REFERENCES

[1] Y.-K.Choi, D.Ha, E.Snow, J.Bokor, T.-J.King, "Reliability study of CMOS FinFETs," in IEEE International Electron Devices Meeting, 2003. IEDM '03 Technical Digest., pp.7.6.1-7.6.4, 8-10 Dec. 2003.

[2] J.Yoshida. "Toyota Case: Single Bit Flip That killed." EETimes, October 2013.

[3] T.Austin, V.Bertacco, S.Mahlke, and Y.Cao. "Reliable Systems on Unreliable Fabrics" IEEE Design & Test of Computers, 25(4): 322-333, July 2008.

[4] H.M.Quinn, A.De Hon, and N..Carter. CCC visioning study: system-level cross-layer cooperation to achieve predictable systems for unpredictable components. Technical report, Los Alamos National Laboratory (LANL), 2011.

[5] S.Mitra, K.Brelsford, and P.N.Sanda. Cross-layer resilience challenges: Metrics and optimization. In Conference on Design Automation & Test in Europe (DATE), 2010.

[6] N.P.Carter, H.Naeimi, and D.S.Gardner. Design Techniques for Cross-Layer Resilience. In Conference on Design, Automation & Test in Europe (DATE), 2010.

[7] A.Vallero, S.Tselonis, N.Foutris, M.Kaliorakis, M.Kooli, A.Savino, G.Politano, A.Bosio, G.Di Natale, D.Gizopoulos, S.Di Carlo, Cross-layer reliability evaluation, moving from the hardware architecture to the system level: A CLERECO EU project overview, Microprocessors and Microsystems, Available online 20 June 2015, ISSN 0141-9331.

[8] M. Ottavi et al. "Dependable Multicore Architectures at Nanoscale: The View From Europe", IEEE Design & Test of Computers, Vol. 32, No 2, Mar.-Apr. 2015, pp. 17-28.

[9] U.Schlichtmann, V.Kleeberger, J.Abraham, A.Evans, C.Gimmler-Dumon, M.Glas, A.Herkersdorf, S.Nassif, and N.Wehn, "Connecting different worlds - technology abstraction for reliability-aware design and test," in Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014, March 2014, pp. 1–8.

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

[11] 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

[12] A.Evans, M.Nicolaidis, S.-J.Wen, D.Alexandrescu, and E.Costenaro, "RIIF - reliability information interchange format," in On-Line Testing Symposium (IOLTS), 2012 IEEE 18th International, June 2012, pp. 103–108.

[13] A.Thomas and K.Pattabiraman, "Error detector placement for soft computation," in Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on, June 2013, pp. 1–12.

[14] A.L.Silburt, A.Evans, A.Burghelea, S.-J.Wen, D.Ward, R.Norrish and D.Hogle. Specification and Verification of Soft Error Performance in Reliable Internet Core Routers. Nuclear Science, IEEE Transactions on, vol. 55, no. 4, pages 2389 –2398, aug. 2008.