Linking Requirements and Design Data for Automated Functional Evaluation

Stephen T. Frezza
Electrical Engineering Dept.
University Square
Gannon University
Erie, PA 16541

Steven P. Levitan
Electrical Engineering Dept.
348 Benedum Hall
University of Pittsburgh
Pittsburgh, PA 15261

Panos K. Chrysanthis
Computer Science Dept.
220 Alumni Hall
University of Pittsburgh
Pittsburgh, PA 15260

Contact Person: Stephen T. Frezza
Phone – (814) 871-7563
Fax – (814) 871-7382
Electrical Engineering Department
Gannon University
University Square
Erie, PA 16541 USA
Email: `frezza@ee.gannon.edu`

All appropriate organizational approvals for the publication of this paper have been obtained. If accepted, the authors will prepare the final manuscript in time for inclusion in the special issue of *Computers in Industry*.

Stephen T. Frezza

Linking Requirements and Design Data for Automated Functional Evaluation

Authors' Biographies

Stephen T. Frezza is an assistant professor in the Department of Electrical Engineering Department at the Gannon University. He received the B.S. degree *cum laude* (1985), the M.S. degree (1991), and the Ph. D. degree (1995) in electrical engineering engineering from the University of Pittsburgh. He worked for Lutron Electronics and Westinghouse Electric Corp. as a project engineer before resuming his education. His research interests include design process modeling, design data management, requirements engineering, and intelligent systems. In 1995, Dr. Frezza joined the Electrical Engineering faculty at Gannon University. Dr. Frezza is a member of the IEEE-CS and the ACM.

Steven P. Levitan is the Wellington C. Carl Associate Professor of Electrical Engineering at the University of Pittsburgh. He received the B.S. degree from Case Western Reserve University (1972) and his M.S. (1979) and Ph.D. (1984), both in Computer Science, from the University of Massachusetts, Amherst. He worked for Xylogic Systems designing hardware for computerized text processing systems and for Digital Equipment Corporation on the Silicon Synthesis project. He was an Assistant Professor from 1984 to 1986 in the Electrical and Computer Engineering Department at the University of Massachusetts. In 1987 he joined the Electrical Engineering faculty at the University of Pittsburgh. Dr. Levitan's research interests include computer aided design for VLSI, parallel computer architecture, parallel algorithm design, and VLSI design. Dr. Levitan is a member of the IEEE-CS, ACM, SPIE, and OSA.

Panos K. Chrysanthis received the B.S. degree in Physics from the University of Athens, Greece, and the M.S. and Ph.D. degrees in Computer and Information Sciences from the University of Massachusetts at Amherst. Currently, he is an Assistant Professor of Computer Science at the University of Pittsburgh. His research interests lie in the areas of distributed database systems and real-time systems. In particular, his research focuses on both formal models and implementation techniques for structuring reliable distributed information systems based on the notions of objects and transactions. Results of his research include the ACTA transaction framework for the analysis and synthesis of extended transactions and workflows found, for example, in CAD/CIM systems and heterogeneous multidatabase systems. Dr. Chrysanthis is a member of the ACM, the IEEE-CS, and Sigma Xi.

Linking Requirements and Design Data for Automated Functional Evaluation

Abstract

This paper presents a methodology for automating the evaluation of complex hierarchical designs using black-box testing techniques. Based on an exploration model for design, this methodology generates evaluation tests using a novel semantic graph data model which captures the relationships between the related design and requirements data. Using these relationships, equivalent tests are generated and systematically applied to simulations of the pieces of a modular design and its requirements. These simulations yield two sets of comparable results, enabling evaluation of partial designs of a complex system early in their design process.

Key Words:

Design Process Modeling	Design Data Management
Traceability	Black-Box Testing

Linking Requirements and Design Data for Automated Functional Evaluation[1]

1 Introduction

The quality of the design for a complex hierarchical system is largely determined by how well it meets the needs and desires for which it was created. These needs, whether implicitly or explicitly stated in a *requirements model*, form the basis upon which the completed system will be judged. The modeled requirements embody these expectations, and the quality of the design suffers when either the requirements differ from the design expectations (poor analysis) or the design does not meet its requirements (poor implementation). Worse, design quality is difficult to assure if the relationships between the requirements and the design are not available to enable comparison. Without a record of these relationships (traditionally referred to as *traceability* [9]), aspects of the design that do not meet the stated requirements are difficult to identify.

Further, the lack of traceability information curtails the effectiveness of design quality evaluation. Conventional design wisdom dictates that quality design is achieved most effectively by evaluating the design early in the design process so that repercussions of changes can be limited [4, 23, 25]. However, without information that relates any independent portion of the design to the requirements for which it was created, early testing cannot take place. Portions of the design, even if completed early cannot be tested against the requirements directly. Therefore, this testing usually waits until a sufficiently large portion of the design, which presumably meets some obvious requirements, is completed.

Hence we have two goals for enhancing design quality:

(1) to develop a database that includes structures for the maintenance of requirements, design, and particularly the traceability information, and

(2) to use these structures to evaluate the quality of the design.

In order to meet the first goal, we develop a unified semantic graph representation of requirements and design data, where the links explicitly represent relations among the requirements and design data. The ability to model the relational links between requirements and design data defines a framework for developing further computer-aided support for concurrent development.

[1]This work was supported, in part, by the National Science Foundation under Grants MIP-9102721 and IRI-9010588. This material in this presentation is partially based on work presented at the 1994 Electronic Design Process Workshop and the 32nd Design Automation Conference [7].

We meet our second goal by developing a methodology that employs this representation to effect automated functional evaluation testing of independent design modules. Successfully automating functional evaluation testing depends on three issues: the modularity of the design, links between design and requirements data, and requirements-based (correct) results for comparison. Modular designs are necessary to enable the use of black-box testing techniques. The links between the design and the requirements enables the correct values from the requirements to be automatically associated with the design variables under test. Correct requirements-based results can either be selected (i.e., literal-value requirements such as timing constraints, etc.) or generated (i.e., simulateable requirements).

In this paper, we present our general methodology for modular hierarchical systems that is based on the above two goals. Throughout this paper we will illustrate the various facets of our approach by applying it to the design of the Floating-Point Arithmetic and Logic Unit (FP ALU) for a DLX 32-bit RISC Microprocessor [10]. This is a modular design based on the requirements for FP functionality as contained in the *ANSI/ IEEE Standard for Binary Floating-Point Arithmetic* [2], and those which can be derived from standard understandings of computer arithmetic [16].

The rest of the paper is organized as follows. Section 2 presents background to this work in the area of design process modeling, particularly the Design As Exploration information process model. Section 3 briefly presents related research in design and requirements representation and introduces our graph model which serves to unify these representations. Section 4 describes a method for automated functional evaluation. Section 5 illustrates the details of how we generate functionality tests for the FP roundoff design. Section 6 summarizes and concludes the paper.

2 Background

The general workflow of the design process can be broadly modeled by: *requirements formulation* + *synthesis* + *analysis* + *evaluation*. When one includes various forms of process feedback, this is a reasonable process model of what designers typically do. This workflow model (with many variations) is common to most cognitive and management models of the design process. Our observation is that this model (implicitly or explicitly) underlies almost all current software and requirements engineering environments, CAD frameworks, and automated design systems.

While cognitive and management models certainly have their uses, our focus is on the organization, structure and use of the design *information*, and not on how a particular set of designers (design agents) manage or order their activities for producing that information. A related view of the design process is to model design as a search process. While search-based models can do an effective job of capturing how design problems are solved, they do not address the interactions between developing

what the design problem is, and finding the solution to that problem. Our work is geared towards a design process model, such as the Design As Exploration (DAE) model [22], that addresses the information interaction between the *development of* and *solution to* the design problem.

The Design As Exploration model, shown in Figure 1, is an attempt to model the nature and characteristics of the process required to solve design problems from a knowledge perspective. This model was intended to serve as a mechanism for developing and expressing how knowledge is organized and how it is used and generated during the process of creating a design. Our main focus in adopting this model is on the need to integrate the information used in design.

(*Typesetters' note: Figure 1 goes here or below...*)

In the DAE model, the design process begins with an identified set of needs or desires that are transformed into an initial requirements description (R_i). Design exploration takes place when changes to one or more of the current requirements (R_c), problem (P_c), or design (D_c) descriptions are made. The current descriptions are then mapped to the final descriptions R_f and D_f. The history of how the P_c,R_c, and D_c descriptions develop is maintained in the design history, H_f, and the three final descriptions comprise the *Design Description Document* (*DDD*). The exploration cycle is continued until P_c forms a well-defined problem statement which embodies all the criteria in R_c and the solution to P_c specified by D_c satisfies R_c. At some point, the design, embodied in the *DDD* actually realizes/satisfies the needs or desires for which the system was created.

The outputs of the exploration process (R_f, D_f, H_f) have two general uses: either they are applied to the upper *learning loop* to extend the knowledge of the problem domain (K_{dm}) and/or the knowledge of how to design in the domain (K_{dn}); or they are applied to the lower *analysis loop* and are analyzed for precision, ambiguity, completeness, correctness, etc.. Results also may be used to create a physical prototype which can serve the same purpose (outer loop). The model does not dictate *how* the design knowledge is created or stored, but rather what kind of knowledge is generated. For example, the analysis loop may include evaluation of mathematical models, generation and use of simulation results, or other evaluation forms or any combination of these techniques.

For our purposes, the DAE model defines two fundamental forms of design data: the *architectural design* (AD) data used to construct a finished device (D_c and D_f) and *requirements definition* (RD) data which represent the requirements model (R_i, R_c and R_f) for the design problem. We utilize the DAE model because it clearly provides for the support of, and the interaction *between* the requirements definition and the architectural design. The DAE model is also useful because it explicitly supports evaluation of the architectural design with respect to the requirements definition. We enhance the DAE model by providing a richer, unified design data model including links between and among the RD and AD data entities. These enhancements permit the practical development of software to help automate a path through the evaluation loop and thus demonstrate the usefulness

of the model.

Recently, a more formal model of the electronic design process has been proposed in [13] which provides a model of sub-problem interaction for the solution of particular design problems. This work parallels the DAE work in that it views the design process as having two fundamental information components, knowledge and data, and provides for the separation and linking of design object desired behavior (RD data) and realization (AD data). However, this work takes a cognitive approach that views design as a search process, and thus does not address the issues involved with simultaneously developing the requirements for *and* the solution to the design problem being solved.

3 A Unified Representation for Requirements and Design

Following the DAE model, our data model takes the view that the critical aspects of data representation are the relationships among and between requirements (RD) and design (AD) data. We use a semantic graph to represent these relationships. We use links in a graph to represent a relationship and maintain the necessary information about the relationship, a feature we rely on heavily for test generation. This is a *unified* approach to design representation, because all of the information and the relationships between different abstraction levels are maintained in the same database [15]. Our unified database has been developed using *ODE* [5], and its O++ language, which is a persistent superset of C++.

3.1 Design Representation

Since design representation work tends to be domain-specific, the AD data representation work for computer hardware has mostly been in the area of VLSI CAD databases. Recent research has focused on the efficient representation and retrieval of design information. A useful example is the *Version Data Model* [14], which explicitly considers *equivalence*, *configuration* and *version* relationships as representational dimensions of design information (see Figure 2a). Configuration relationships support the design hierarchy, equivalence relationships describe how one design description is similar to another design description for the same design object, and version relationships describe how one variant of a design entity is related to another variant of the same entity.

(*Typesetters' note: Figure 2 goes here or below...*)

An example of a portion of the DLX FP ALU design captured in our unified data model is shown in Figure 3. This figure shows the configuration links that trace the hierarchy of the VHDL (VHSIC Hardware Description Language) [11] entities for the ALU, as well as several version links from the *Add-to-infinite-precision* entity, with the non-current versions linked to the current version. While the versions of this entity are all shown linked to the current entity, they could also be arranged in a

tree-like fashion. Figure 3 also contains three equivalent representations for the *FP÷* (floating-point division) entity: one in each VHDL, Magic [20], and Spice [19]. Each of these entities are connected by equivalence links, indicating that the representations for the entity are functionally equivalent. For the *FP÷* AD entity, we show a Spice netlist derived from a Magic layout which in turn has been derived from the VHDL architecture/entity pair. Our prototype system supports VHDL design entities.

(*Typesetters' note: Figure 3 goes here or below...*)

3.2 Requirements Representation

Most requirements representation work has focused on the development of requirements frameworks, e.g., [6]. Comparing the design data model and these frameworks, we identify a similar set of relationships that exists for requirements data, as shown in Figure 2b. Requirements entities include the three types of relationships that design entities have: *configuration* (is-part-of), *equivalence* (same-as) and *version* (derived-from). In addition, our requirements model supports *viewpoint* (related-to) relationships, which are used to distinguish different stakeholder's views on the requirements of the system being developed. This data model allows requirements information to be stored using different description types, such as entity-relationship [6] or line-item [1], and allows representations for the connections among the RD entities.

Figure 4 shows an example of RD in our unified data model, representing some line-item requirements and their equivalent simulateable representations. Shown are fragments of the requirements for floating-point number formats required for the FP ALU employed in the DLX. The *ANSI/ IEEE FP Standard*, the required FP number *Formats*, as well as the particular *Sets of Values*, are depicted at the top right-hand corner of the figure. Each of the number formats take on particular *Sets of Values*; for which *Precision*, *Max* and *Min Exponent* values, etc., are defined.

(*Typesetters' note: Figure 4 goes here or below...*)

Also shown are the *Basic Formats* requirements which are comprised of configuration links to field definitions and a set of equivalence links to formal requirements constructs. The equivalence relations link simulateable specification statements to particular requirements, e.g., `DATA: SP_EXPONENT_OFFSET`, is associated with the *Exponent Bias=127* requirement. Figure 4 depicts one viewpoint, but no version relations.

We are particularly interested in *simulateable requirements* in order to be effective in generating functional evaluation tests. Different simulateable requirement modeling languages have been proposed [24], and the simulateable requirements modeling language we have integrated into our system is the *Requirements Specification Language* (RSL) [1]. RSL's availability, its ability to support in-line requirements and its simulation semantics make it a useful example of the type of requirements

modeling languages that our unified data model can support.

RSL has its limitations in that it belongs to a class of system-specification languages whose improper use can lead to low flexibility in the design, as the inherent over-specification obstructs change [17]. However, RSL has reasonable simulation semantics which we employ to demonstrate the value of being able to incrementally test the implementation of a developing design against the requirements model.

3.3 Linking the Requirements and the Design

A key aspect to being able to use any simulateable requirements model effectively is the ability to focus the simulation on the appropriate part of the requirements model. Here the relational links within and between the RD and AD data classes serve a key role by providing the means for identifying the subset of requirements applicable to the design module under consideration. When properly constructed, these relational links are termed *traceability links* because they provide a thread of origin from the implementation to the requirements [9], and serve as a validation that the design does indeed do what it was intended to do.

We identify and support four categories of traceability links, and identify their roles in tracing through the RD and AD data. Two of these categories represent *intra*-dependencies, that is, dependencies within the RD and AD data classes. Similarly, two categories represent *inter*-dependencies, that is, the dependencies between the design data classes. For example, [RD→AD] denotes the link types that indicate how some AD data is dependent upon some RD data.

Rational Dependency [$RD \rightarrow AD$]: purpose of design object is tied to a particular (non-null) set of requirements; normally called forward traceability links [21].

Technical Dependency [$AD \rightarrow AD$]: dependence of one design object on another to perform/meet its requirements. This link relates to the design entities' combined ability to do the right thing - and typically encompasses the interface/connections internal to the design. These links also include the configuration relations among the design entities.

Contextual Dependency [$RD \rightarrow RD$]: purpose of requirement object is tied to other requirements objects, and includes the configuration relations among RD data. Can include requirements that are derived (or implicitly stated) in the environment, such as where optative descriptions imply (or rely upon) assertive descriptions within the requirements model [12].

Implicative Dependency [$AD \rightarrow RD$]: assertion of a design entity implies other requirements/constraints on the design. A typical example would be where design decisions

affect/influence the requirements definitions. Similar to contextual dependencies, these form one class of links normally called reverse traceability links [21].

(*Typesetters' note: Figure 5 goes here or below...*)

Figure 5 illustrates these four types of links. Several rational dependencies are shown, e.g., the link from the *FP+_Completion* RD entity to the *FP+* AD entity. This link identifies how, for the *FP+* (Floating-Point addition) design to succeed, it must address the *FP+_Completion* requirement. Technical dependencies are shown linking the *SP_Format* (Single-Precision Format) and *DP_Format* (Double-Precision Format) AD entities to the *Unpack* AD entity, indicating that the operation of *Unpack* depends upon the implementation details of the two format entities. Contextual dependencies are shown linking the *Arithmetic* RD entity to the *Supported_Operations* and *FP+_Speed* RD entities. This contextual link captures the notion that details of the arithmetic requirements are addressed by the *Supported_Operations* and *FP+_Speed* RD entities. Finally, an implicative dependency is shown linking the *DP_Format* AD entity and the *Support_SP_and_DP_only* RD entity. This is an implicative dependency because the *DP_Format* is a design decision, and is not explicitly required for the DLX. While rooted in the requirements definition for the DLX, the *DP_Format* influences the requirements that relate to the support required for all FP number formats.

To summarize, our semantic graph model is unique in that it identifies the classes of relationships that need to be maintained *within* and *between* the requirements and design data. In each part of the model, we use links to both represent a relationship and maintain the necessary information about the relationship. One key benefit of the model is highlighted in the next section, where we illustrate the ability to automatically generate black-box functional evaluation tests.

4 Automated Functional Evaluation

Functional evaluation testing answers the question, "does a particular part of the design function correctly?" We approximate 'correct' behavior by simulating the requirements associated with the design entity in question. Since we focus on evaluating a piece of the design with respect to an identified set of requirements and not with its internal workings, black-box testing techniques are most appropriate.

We implement black-box functional evaluation testing by employing boundary-value analysis and equivalence partitioning techniques [18]. While this is not the only test-case design strategy available (or even necessarily the most effective [3]), it shows promise of being able to uncover many errors at a reasonable cost, where cost is the number of test runs per error discovered. In general, this type of black-box testing involves the generation of a black-box *testset*, the application of this testset to generate both 'correct' (requirements simulation) and implemented (design simulation) results, and

finally a comparison of these two sets of results. Our focus is on automating one such black-box test-generation technique for requirements-based tests.

This form of test-generation presupposes the existence of a simulateable requirements representation, a simulateable design representation and input classes that map equivalently to both the requirements and the design simulations. In our case, the testset generation and application process is divided into six steps.

The first step is to generate a simulateable requirements specification (SRS) for the design entity under consideration. As the design is not necessarily directly traceable to the simulateable requirements, Figure 6 shows the three substeps involved: Tracing to the set of related requirements (**1a**), tracing to the subset of simulateable (RSL) requirements (**1b**), and constructing the SRS (**1c**).

(*Typesetters' note: Figure 6 goes here or below...*)

The second step is to create an I/O Specification based on the names that will be used in each of the requirements and design simulations (**2**). We use the semantic link information for insuring that the I/O specification includes the AD names (or fields) that relate to the RD names that will be used in the testset. This is a key factor for ensuring that the test results generated from each simulation are directly comparable.

Effective black-box testing depends on tailoring the testset to the design entity under consideration. We use information from the (design) I/O specification and the SRS to generate a testset (**3**) tailored to the design entity. This involves identifying the input classes from the SRS and I/O specification and selecting appropriate boundary values for each input class, thus the class names map from the SRS (RD), and parallel the design (AD) names. Input classes consist of input ranges, determined by the data type used in the design.

Simple heuristics based on the RD types are applied to the input ranges of each class to determine what values to test for. E.g., for bit strings the min, min+1, mid, max-1 and max values are tried; for enumeration types, all enumeration values are used. These combinations are then checked for redundancies in order to keep the generated testset from growing unnecessarily large. Other heuristics could be applied as well.

Figure 7 depicts the application of the generated, design-specific testset for performance evaluation. The fourth step in the process is to apply the testset to the SRS to generate the 'correct' results (**4**). Similarly, the fifth step is to apply the testset to a simulation of the implemented design to generate the implementation results (**5**). Since we provide for a data model that supports many forms of design and requirement representations, applying a testset to a particular representation involves selecting/generating a simulator for the representation and mapping the testset values to the inputs to the simulation. For example, RSL requires a custom simulator for each SRS, whereas VHDL

might have two types of simulators (behavioral and structural) depending on the design.

(*Typesetters' note: Figure 7 goes here or below...*)

The last step of the process is to compare the simulation results to the correct results (**6**). The presentation and comparison of the test results is important, as all discrepancies need to be highlighted, and the individual tests made available to the designer. We do not address the interface issues, as our emphasis is on generating the information rather than presentation.

Our functional evaluation methodology is effective if either the requirements linked to the design trace to a set of simulateable requirements or to explicit functional values (e.g., measurable constraints like time, value ranges) where the latter is a topic of future work. In our prototype implementation [8], we linked the design variables directly to their (respective) simulateable requirements. However, in the general case design variables could be linked to *any* requirement, and graph-tracing algorithms could be employed to trace from any requirement to its derivative simulateable requirements.

This methodology focuses on the evaluation of partial designs to enable the use of requirements-based testing early in the design process. However, what constitutes a partial design is a process-management issue. A partial design need only have a clearly defined module interface, but may in fact consist of several (already tested) partial designs. The potential exists for exponential growth in the size of the testset(s) due to growth in the interface to the (now larger) partial design. However, if the designers use proper modularity in defining their partial designs, then much of this problem is alleviated, and the testsets for the larger partial design will serve as a form of integration testing, and validate the sum of the parts.

Similarly, *when* automated functional testing should occur is also a process-specific (management) issue. Our work shows that automated functional evaluation testing of a partial design is possible, given simulateable requirements, a modular design, and links between design and requirements data.

5 Roundoff Example

In this section, we illustrate the evaluation of an isolated part of a design using our methodology. Returning to our example of a floating point arithmetic and logic unit (FP ALU), consider the implementation of roundoff, a small yet important aspect of the design of the FP ALU in the DLX. The design of our FP Adder was broken down into six stages, of which the roundoff was only one: The other five are: unpack, pre-normalization, add-to-infinite-precision, post-normalization, and post-result (as shown in Figure 3). Our goal is to evaluate the roundoff portion of the design, independent of the rest of the design. For clarity, we present a brief description of FP adder operation.

(*Typesetters' note: Figure 8 goes here or below...*)

The two FP numbers at the top of Figure 8 are shown represented in the IEEE single-precision format *Before Unpacking*. In the *Pre-Normalization* stage, they are converted to have a signed exponent, and the smaller of the two numbers is shifted to have the same exponent as the larger. When added, these two pre-normalized numbers create an *Add-to-infinite-precision* sub-result that will cause the *Roundoff* stage to increment the exponent. In *Post Normalization*, the value is checked, and the corrected exponent is converted from signed to offset convention. Here we assume the default Round-To Nearest Even mode. Finally, the leading bit of the mantissa is trimmed, and the *Result Posted*.

Our focus is on the design of the roundoff stage for single-precision values in all four mandated roundoff modes (Round-to-nearest-even, Round-to-positive-infinity, Round-to-negative-infinity, and Round-to-zero). Because we want to evaluate the roundoff design early in the design process, we want to test it independently of the rest of the design. To this end, we follow the testset generation procedure (Figure 6) to generate a roundoff SRS and I/O specification from the linked RD data, and use this information to generate a testset - yielding six input classes.

The input class names for the roundoff testset come from the linked requirements entities: *mode, sign, exponent, round, fraction*, and *sticky*. The data types used to define the ranges for the black-box input classes come from the the corresponding I/O specification variables: *Mode(1 downto 0), SPostNorm, EPostNorm(7 downto 0), MPostNorm(0), MPostNorm(23 downto 1)* and *StickyPN* respectively. Before testset reduction, these six input classes would each have five potential values: Min, Min+1, Mid, Max-1, and Max. For example, the VHDL designer represented the post-normalized exponent field (*EPostNorm*) as an eight-bit value, which would be mapped to the five values: `00000000`, `00000001`, `01111111`, `11111110`, and `11111111`.

Without testset reduction, this technique would yield $5^6 = 15625$ test cases. However, by using the RD type information accessible via the traceability link, we can determine that *mode* is a control variable and is associated with the four required FP rounding modes, *sign* is a single bit having exactly two values, *exponent* is a string of bits which can take on the five specified test values, *round* has exactly two values, *fraction* takes on five values, and *sticky* has exactly two values. This yields a testset containing $4 \times 2 \times 5 \times 2 \times 5 \times 2 = 800$ cases.

These input classes and their corresponding values and ranges are shown in Table 1 which summarizes the reduced testset for *round*. The values depicted were used to generate a set of equivalent SRS and design simulation inputs, which in turn were used to calculate the roundoff requirements simulation (correct) and design simulation results.

(*Typesetters' note: Table 1 goes here or below...*)

Figure 9 presents parts of the test results for the VHDL design entity *round*. Specifically, the figure depicts several test cases (identified by the unique min/max test value combinations) for the required

rounding mode RTNE (Round To Nearest Even). The design simulation output variable is `round`, and the correct results are shown in the `correct round = ...` output line.

(*Typesetters' note: Figure 9 goes here or below...*)

The summary results for test cases 19 and 20 of Figure 9 show discrepancies between the correct results (`correct round = ...`) and the implemented results (`round = ...`). As it turns out, these two errors were caused by an incorrect exponent increment case in the original behavioral design for *roundoff*. This extraneous code was determined to be the cause of the four other errors detected by the 800-case testset (test cases 59, 60, 159, and 160, not shown here).

Once the extraneous code was removed, subsequent use of the testset discovered no more errors. Here the discovery of the initial error, its correction, and subsequent regression testing was successful in establishing confidence in the roundoff design independent of the completion of the rest of the ALU design. By enabling the discovery of discrepancies early in the design process, this example demonstrates (on a small scale) how our requirements-based evaluation methodology can be effective in improving the quality of a design.

6 Summary and Conclusions

In this paper, we have presented an information process model for the design of complex hierarchical systems supported by a unified semantic graph representation that links requirments and design data. Based on this data model, we presented a methodology for automating functional evaluation testing of complex hierarchical systems in an incremental and modular fashion using black-box testing techniques. We also presented the details of an example showing the generation of black-box functional tests for the roundoff of a floating-point adder.

The semantic graph data model presented successfully supports requirements, design and traceability information. Since the semantic graph was implemented in a object-oriented database it can support large designs with numerous relations. Further, the use of an OODB representation of the semantic model enabled the automation of a functional evaluation methodology for partial designs of complex hierarchical systems. The functional evaluation methodology presented is both useful in identifying design errors and is practical to implement. The methodology is also scalable, that is it uses hierarchy to keep the size of the tests manageable. The methodology is dependent on having both the evolving requirements and design data in the database. While providing the means to capture this information in the database is itself non-trivial, we have shown the benefits of this effort in effecting quality results in an dynamic design process environment.

References

[1] Alford, M. Software requirements engineering methodology (SREM) at the age of eleven: Requirements driven design. In *Modern Software Engineering: Foundations and Current Perspectives*, chapter 11, pages 351–377. Van Nostrand Reinhold, 1990.

[2] American National Standards Institute and the IEEE Standards Board, New York, NY. *IEEE Standard for Binary Floating-Point Arithmetic*, 1985. ANSI/IEEE Std 754-1985.

[3] Beizer, B. *Black-Box Testing.* John Wiley & Sons, 1995.

[4] Brooks, Jr., F. P. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[5] Dar, S., Gehani, N., and Jagadish, H. ODE object database and environment. Technical report, AT&T Bell Laboratories, CTR Lab, 1991.

[6] Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., and Goedicke, M. Viewpoints: A framework for integrating multiple perspectives in system development. *Int. J. of Software Engineering and Knowledge Engineering*, 2(1):31–58, March 1992.

[7] Frezza, S., Levitan, S., and Chrysanthis, P. Requirements-Based Functional Evaluation. *32nd ACM/IEEE Design Automation Conf. (DAC '95)*, pages 76–81, June 1995.

[8] Frezza, S. T. *Requirements Based Design Evaluation Methodology.* PhD thesis, Department of Electrical Engineering, University of Pittsburgh, Pittsburgh, Pennsylvania, 1995.

[9] Gotel, O. and Finkelstein, A. Modeling the contribution structure underlying requirements. *Proc. of the 1st Int. Conf. on Requirements Engineering (ICRE)*, pages 94–101, April 1994.

[10] Hennessey, J. L. and Patterson, D. A. *Computer Architecture a Quantitatitve Approach.* Morgan Kaufmann, 1990.

[11] IEEE Press, New York, NY. *IEEE Standard VHDL Language Reference Manual*, 1993. IEEE Std 1076-1993.

[12] Jackson, M. and Zave, P. Domain descriptions. In *Proc. of the IEEE Int. Symp. on Requirements Engineering*, pages 56–64, January 1993.

[13] Jacome, M. and Director, S. A formal basis for design process planning and management. In *IEEE/ACM Int. Conf. on CAD-94*, pages 516–521, November 1994.

[14] Katz, R. H. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, December 1990.

[15] Kollaritsch, P., Lusky, S., Matzke, D., Smith, D., and Stanford, P. A unified design representation can work. In *26th ACM/IEEE Design Automation Conf.*, pages 811–813, June 1989.

[16] Koren, I. *Computer Arithmetic Algorithms.* Prentice Hall, 1993.

[17] McDermid, J. A. *Requirements Engineering - Social and Technical Issues*, chapter Requirements analysis: Orthodoxy, fundamentalism and heresy, pages 17–40. Academic Press, 1994.

[18] Myers, G. J. *The Art of Software Testing.* John Wiley & Sons, 1979.

[19] Nagel, L. W. Spice2: A computer program to simulate semiconductor circuits. Technical Report ERL-M520, University of California at Berkeley, May 1975.

[20] Ousterhout, J. K. Cornerstiching: A data-structuring technique for vlsi layout tools. *IEEE Transactions on Comp. Aided Design*, CAD-3(1):87–100, 1984.

[21] Ramesh, B. and Edwards, M. Issues in the development of a requirements traceability model. In *Proc. of the IEEE Int. Symp. on Requirements Engineering*, pages 256–259, January 1993.

[22] Smithers, T. Design as exploration: Puzzle-making and puzzle-solving. In *Workshop on Search-Based and Exploration-Based Models of Design Process*, pages 1–21, June 1992. Held in conjunction with 2nd Int. Conf. on Artificial Intelligence in Design.

[23] Sommerville, I. *Software Engineering.* Addison-Wesley, 3rd edition, 1989.

[24] Webster, D. E. Mapping the design information representation terrain. *IEEE Computer*, 21(12):8–23, December 1988.

[25] Weller, E. F. Using metrics to manage software projects. *IEEE Computer*, 27(9):27–33, September 1994.

DL

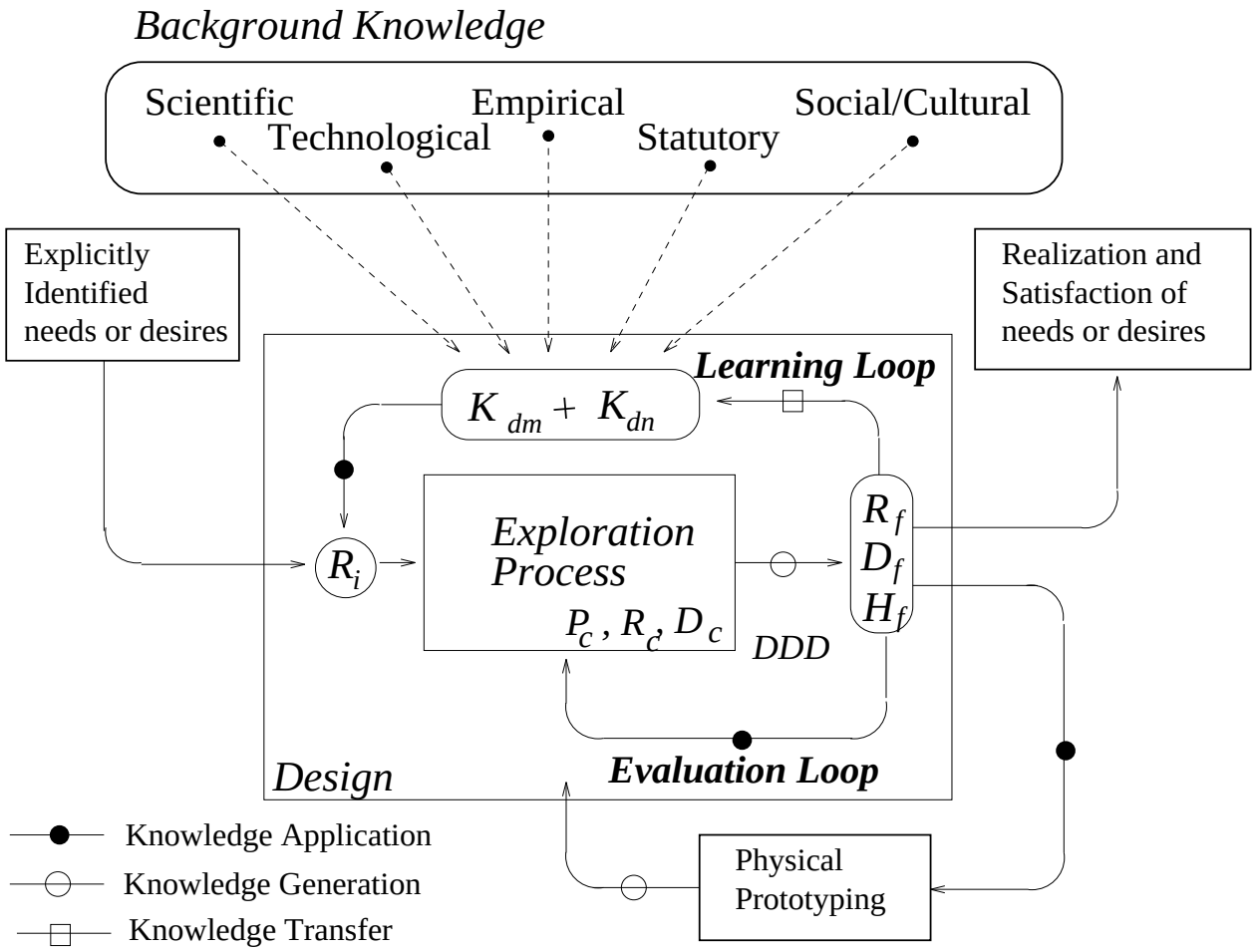

Figure 1: Design As Exploration (DAE) Information Process Model

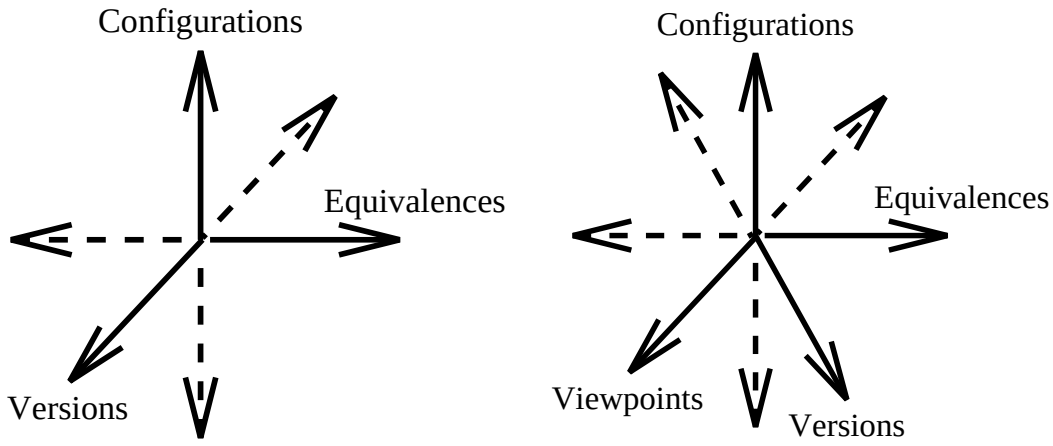

Figure 2: (a) Version Data Model (AD) (b) Requirements Data Model (RD)

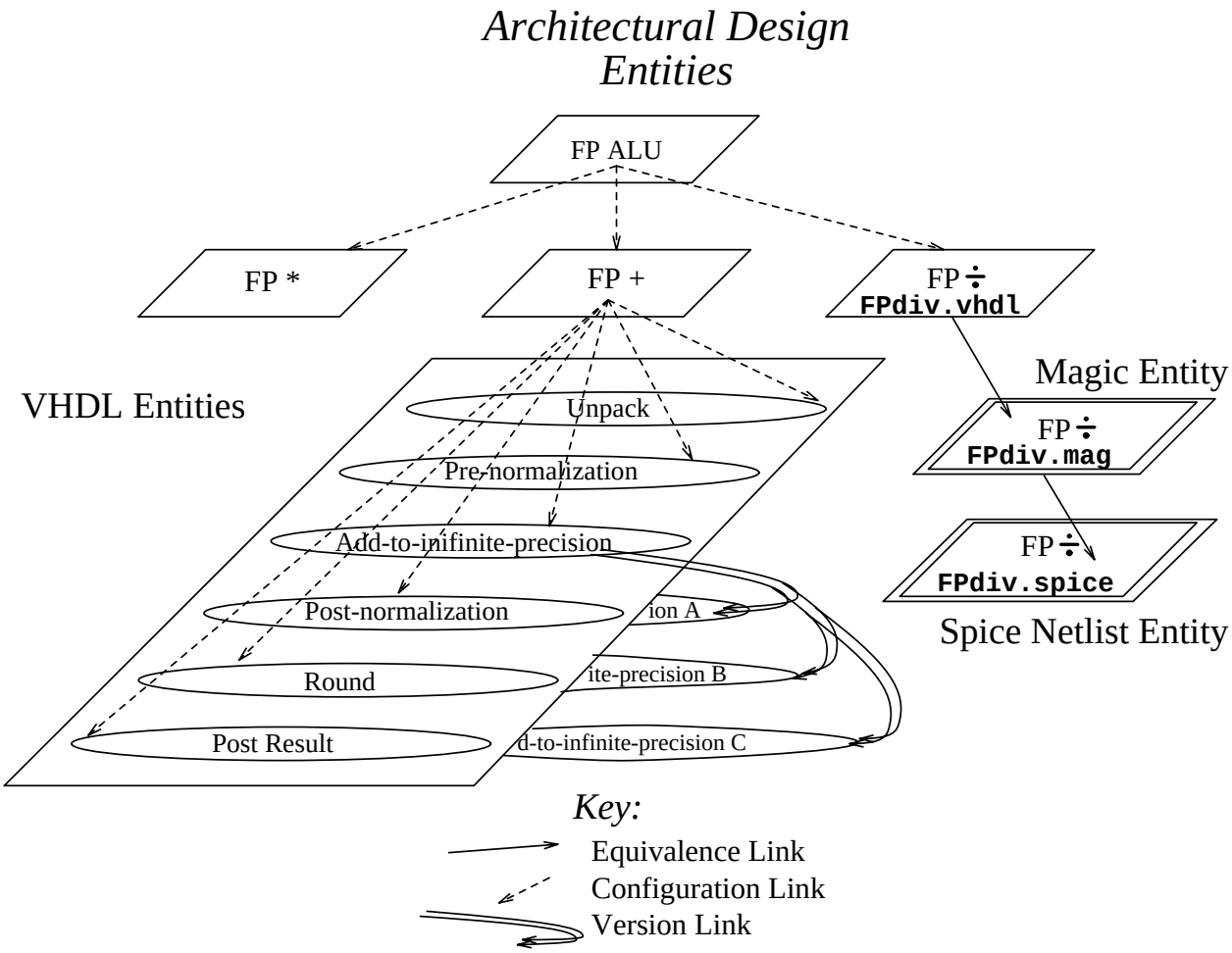

Figure 3: Different views of the DLX floating-point ALU design

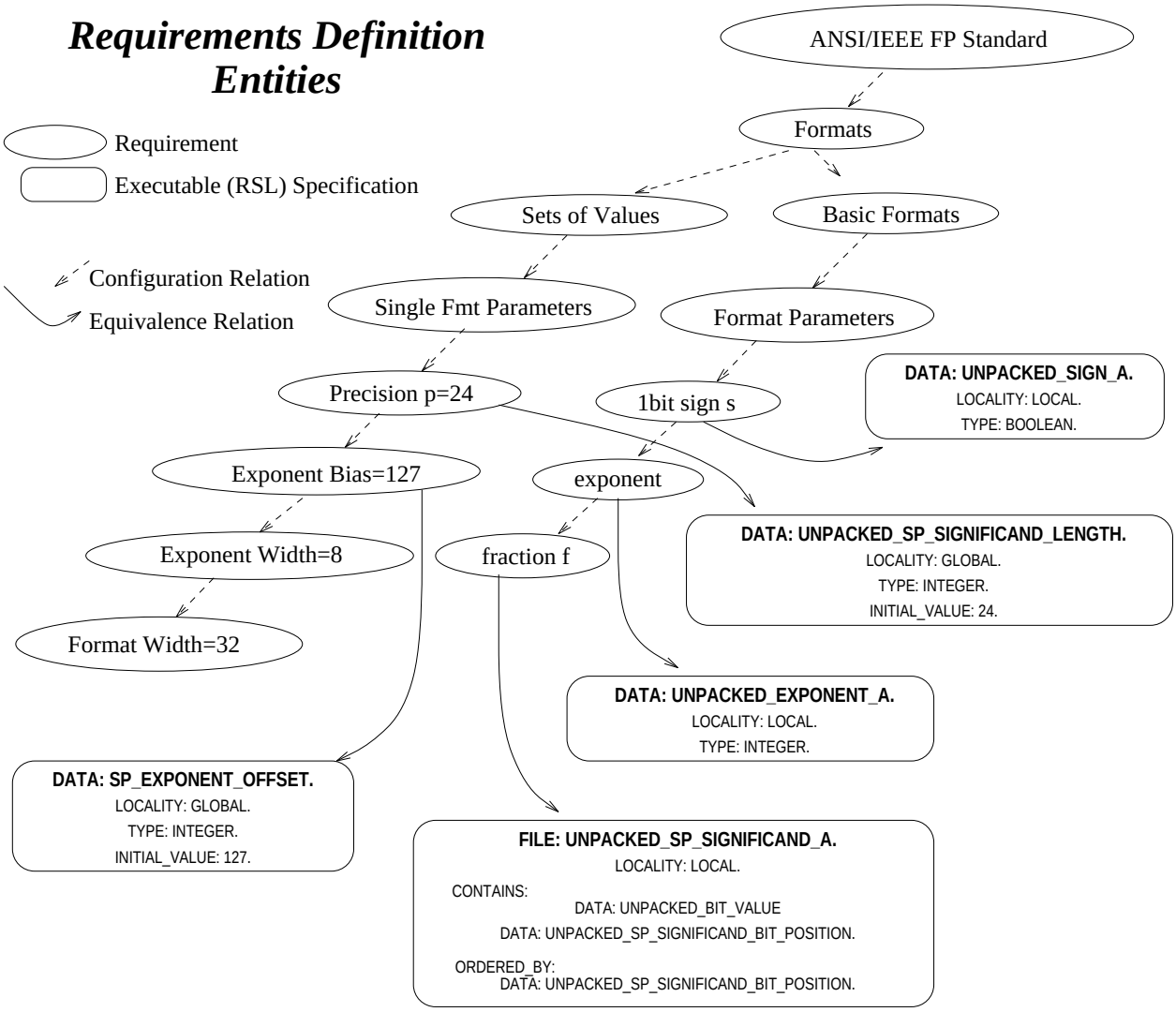

Figure 4: Example of a requirements hierarchy showing configuration and equivalence relations

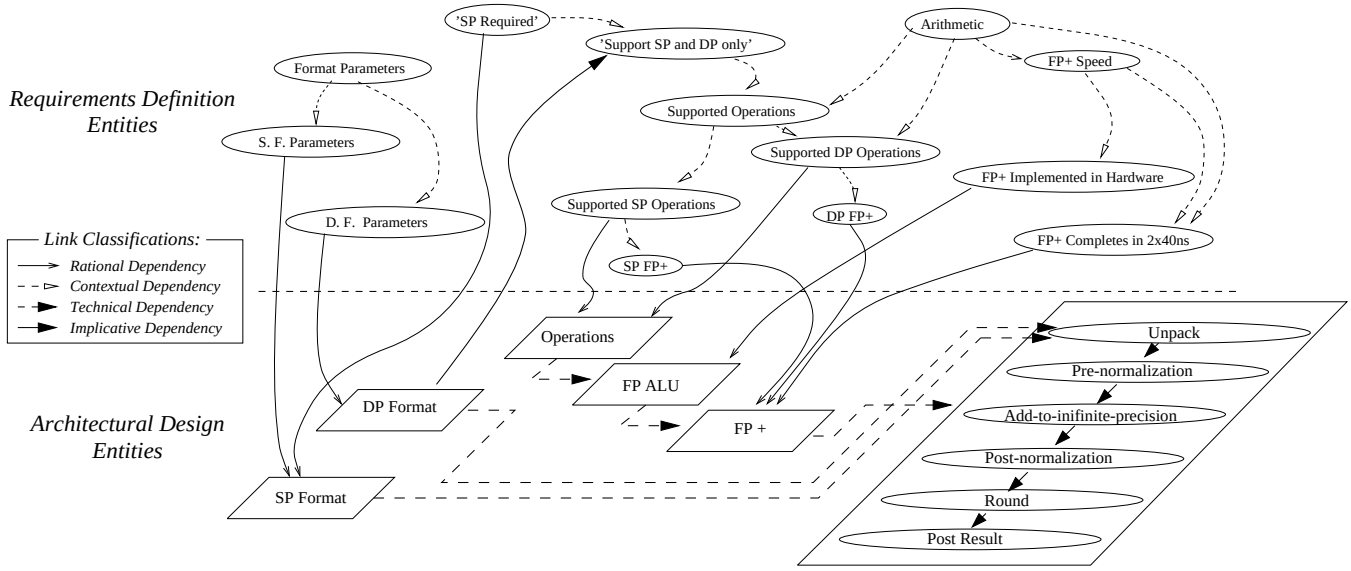

Figure 5: Linked Requirements and Design Entities

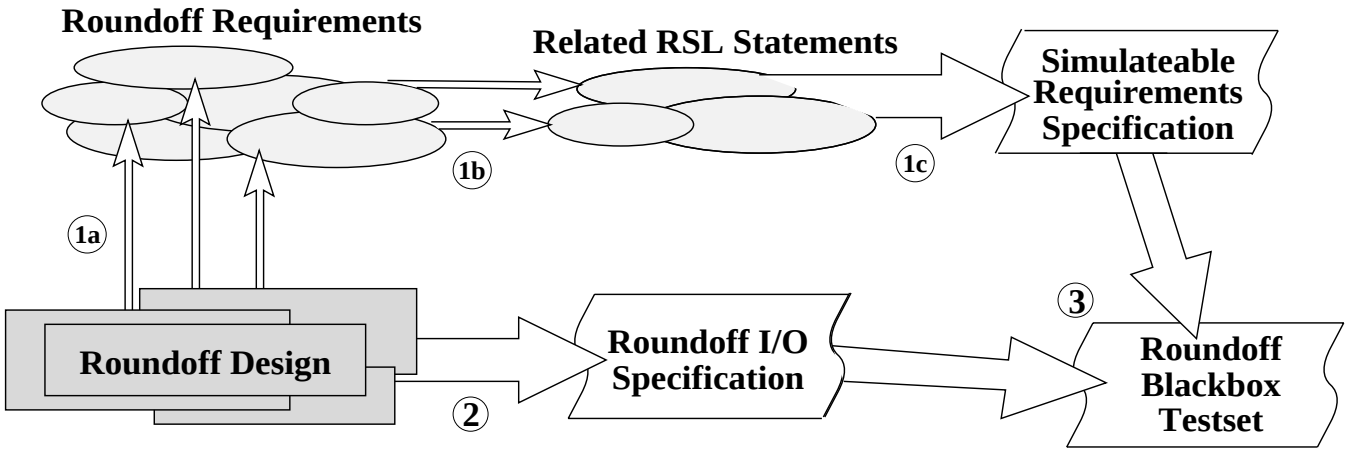

Figure 6: Black-box testset generation for FP ALU roundoff

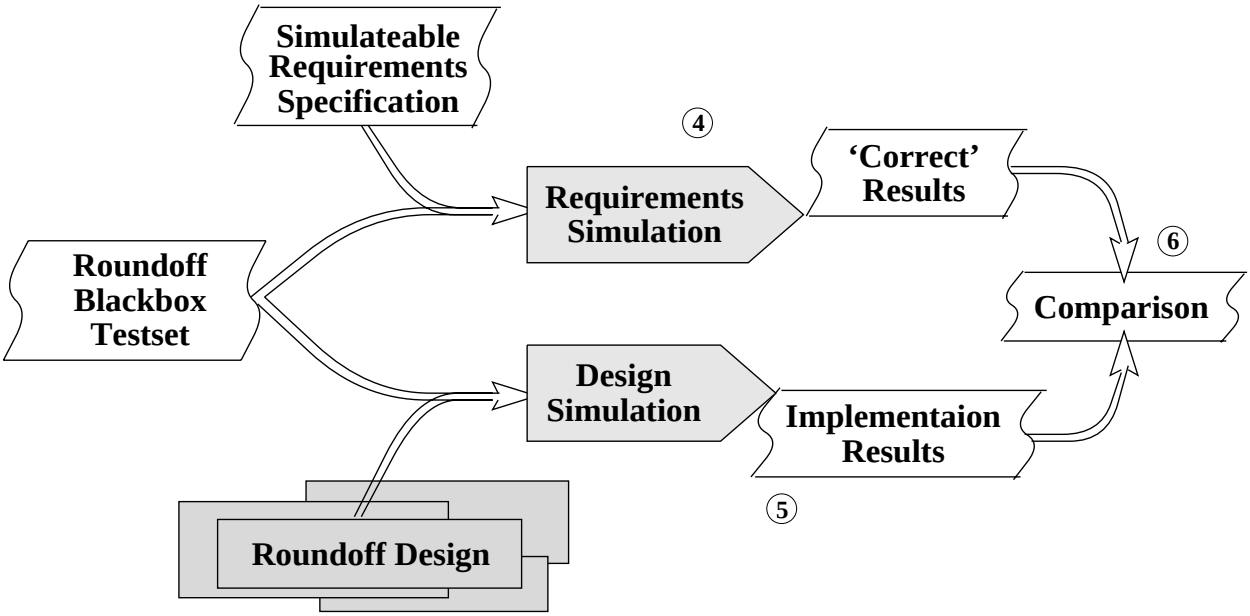

Figure 7: Roundoff testset application for performance evaluation

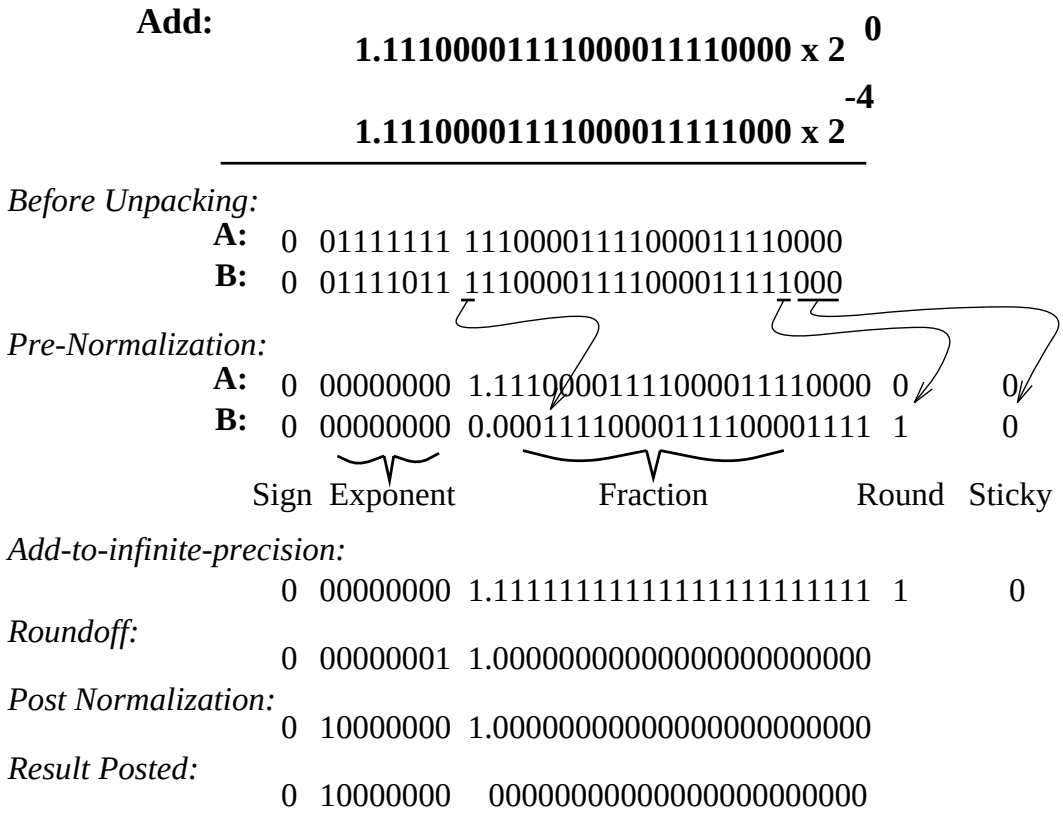

Figure 8: FP Addition with Round-to-nearest-even

Class name [design name]	Values	Count	Notes
Mode [Mode]	00, 01, 10, 11	4	Enumerate controls
Sign [SPostNorm]	0, 1	2	Reduces to Min/Max
Exponent [EPostNorm]	00000000, 00000001, 01111111, 11111110, 11111111	5	Min, Min + 1 Mid Max - 1, Max
Round [MPostNorm(0)]	0, 1	2	
Fraction [MPostNorm (23..1)]	11111111111111111111111, 11111111111111111111110, 01111111111111111111111, 00000000000000000000001, 00000000000000000000000	5	Max Max - 1 Mid Min + 1 Min
Sticky [StickyPN]	0, 1	2	

Table 1: Reduced input test classes and values for FP roundoff

```
  Test     mode   sign   expn   frac   round  sticky
|=================================================
| Case 18  RTNE    min    min    max    min    max
| correct round = 00000000011111111111111111111111
|  round        = 00000000011111111111111111111111
|  time = 36000 = 3600.0ns
| Case 19  RTNE    min    min    max    max    min
| correct round = 00000000100000000000000000000000
|  round        = 00001100100000000000000000000000
|  time = 38000 = 3800.0ns
| Case 20  RTNE    min    min    max    max    max
| correct round = 00000000100000000000000000000000
|  round        = 00001100100000000000000000000000
|  time = 40000 = 4000.0ns
```

Figure 9: Summary black-box test results for FP ALU roundoff showing two detected errors