Requirements-Based Design Evaluation[1]

Stephen T. Frezza Steven P. Levitan
Dept. of Electrical Engineering
University of Pittsburgh
{frezza, levitan}@ee.pitt.edu

Panos K. Chrysanthis
Computer Science Department
University of Pittsburgh
panos@cs.pitt.edu

Abstract— This paper presents a methodology for automating the evaluation of partial designs using black-box testing techniques. This methodology generates black-box evaluation tests using a novel semantic graph data model to maintain the relationships between the related design and requirements data. We demonstrate the utility of this technique by using the relationship information to automatically generate and run functionality tests of partial designs against the related requirements.

I. Introduction

Current Computer-Aided Design (CAD) tools are geared to help the designer develop good designs from specifications. Likewise, current computer-aided requirements tools are geared to help the analyst formulate good requirements, and create specifications. However, designs often have their requirements change while the design is being developed, and there is little support for continually reflecting these changes in the specifications. Consequently, the quality of the design (i.e., how well the design meets the expectations placed on it) suffers.

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* [5]), aspects of the design that do not meet the stated requirements are difficult to identify. Hence our goal is twofold: (1) to develop a design system where the requirements, design, and particularly the traceability information can be maintained, and (2) to use these structures to help evaluate the quality of the design.

In order to meet the first part of this 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. We then demonstrate the usefulness of this effort by developing a means of testing the design quality by using trace information to automate functional evaluation testing of design modules.

[1] This work was supported, in part, by the National Science Foundation under Grants MIP-9102721 and IRI-9010588.

The rest of the paper is organized as follows. Section II presents background to this work in the area of design data and process modeling, along with related research in design and requirements representation. Section III introduces our graph model, which unifies requirements and design representations. Section IV describes a method for automated functional evaluation. Section V illustrates the details of how we generate functionality tests for the roundoff design of a floating-point Arithmetic and Logic Unit (FP ALU). This section discusses the testset results for the roundoff example. Section VI concludes the paper with a summary.

II. Background

The general flow 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 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 design process is to model design as a search process. While search-based models can do an effective job of modeling 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 gears towards a design process model, such as the Design As Exploration (DAE) model [15], that addresses the information interaction between the *development of* and *solution to* the design problem.

The DAE model defines two fundamental forms of design data: the *architectural design* (AD) data used to construct a finished device and *requirements definition* (RD) data which represent the requirements model for the design problem. In the model, the RD and the AD are developed simultaneously, in a process termed *design exploration*. The DAE model incorporates two forms of knowledge feedback: design learning and evaluation. 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.

Recently, a more formal model of the electronic design process has been proposed in [8] which provides an excellent model of sub-problem interaction for the solution of particular design problems. This model 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, the model uses a cognitive model 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.

Design and Requirements Data Representation

Following the DAE model, our enhanced model takes the view that there are two broad classes of design information: RD data and AD data. This view runs contrary to *orthodox* requirements engineering, where different forms of specifications are often considered separately from the requirements definition. Our *heretical* [12] view is that specifications are combinations of design and requirements data that serve specific (e.g., contractual or descriptive) purposes, and that the critical areas of data representation and relation are those of the various forms of requirements and design data.

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, particularly version and equivalence maintenance. For example, in the *Version Data Model* [9], the *equivalence*, *configuration* and *version* relationships are explicitly considered as representational dimensions of design information (see Figure 1a). 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.

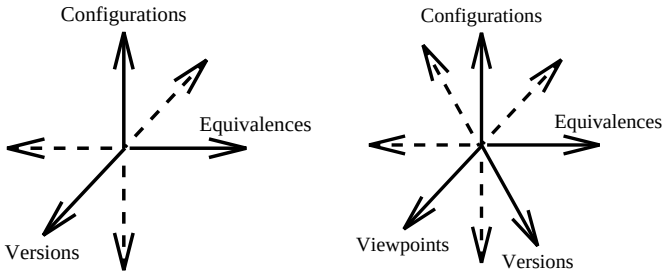

Fig. 1. (a) Version Data Model (AD) (b) Requirements Data Model (RD)

The Version Data Model is interesting in that it represents a common terminology and collection of mechanisms for representing engineering design data. However, like most of the CAD database work, this model fails to address the need to provide for and maintain the relationships *between* requirements and design data.

Most requirements representation work has focused on the development of requirements frameworks, e.g., [4]. Comparing the design data model and these frameworks, we identify a similar set of relationships that exists for requirements data. Our requirements data model is shown in Figure 1b. Similar to the Version Data Model, 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 for aspects of the system being developed. This data model allows requirements information to be stored using different description types, such as entity-relationship [4] or line-item [1], and allows representations for the connections among the RD entities.

III. A Unified Representation for Requirements and Design

The semantic graph data model introduced by our work is unique in that it identifies the classes of relationships that need to be maintained *between* the requirements and design data. In each part of the model, we use links 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 [10].

A. Design Representation

An example of AD captured in our unified data model is shown in Figure 2. Here we introduce a part of the Floating-Point Arithmetic and Logic Unit (FP ALU) for a DLX 32-bit RISC Microprocessor [6]. The design is 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 [11]. We discuss aspects of the FP ALU example in more detail in Sections IV and V.

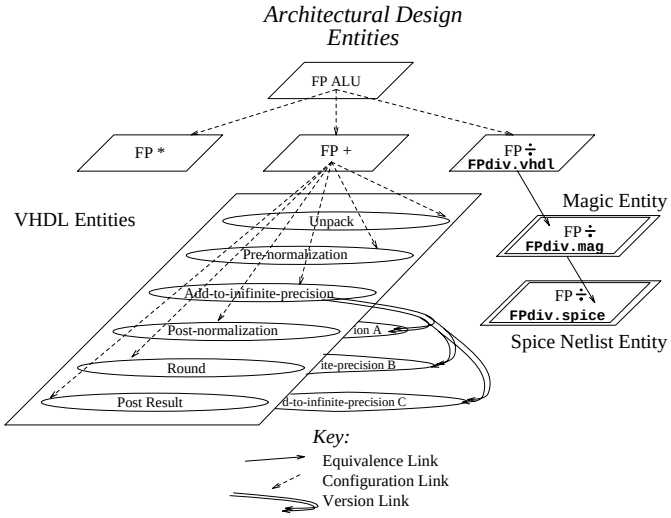

Fig. 2. Different views of the DLX floating-point ALU design

This figure shows a part of the VHDL view of the FP ALU. Shown are configuration links that trace the hierar-

chy of the VHDL entities, 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 2 also contains three equivalent representations for the *FP÷* entity: one in VHDL, Magic, and Spice. Each of these entities are connected by equivalence links, indicating that the representations for the entity are circuit-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.

B. Requirements Representation

Figure 3 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.

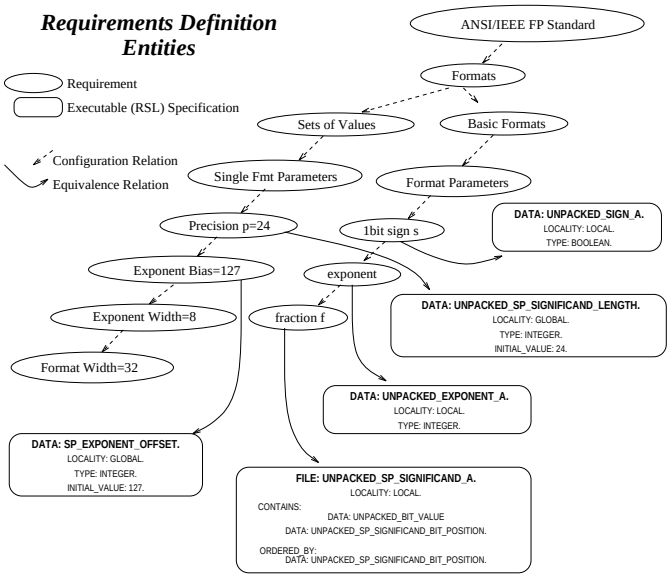

Fig. 3. Example of a requirements hierarchy showing configuration and equivalence relations

Also shown are the *Basic Formats* requirements which are comprised of configuration links to field definitions and a set of equivalence relations to RSL 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 3 shows one viewpoint, and no no versions are shown.

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 [16], 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 inline requirements as well as simulation semantics makes it a useful example of the type of requirements modeling languages that our unified data model can support.

RSL has its limitations in that orthodox system-specification languages like RSL can lead to low flexibility in the design as the inherent over-specification obstructs change [12]. 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.

C. 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 [5], 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→AD*]: purpose of design object is tied to a particular (non-null) set of requirements; normally called forward traceability links [14].

Technical Dependency [*AD→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→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 [7].

Implicative Dependency [*AD→RD*]: 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 [14].

Figure 4 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+* design to succeed, it must address the *FP+_Completion* requirement. Technical dependencies are shown linking the *SP_Format* and

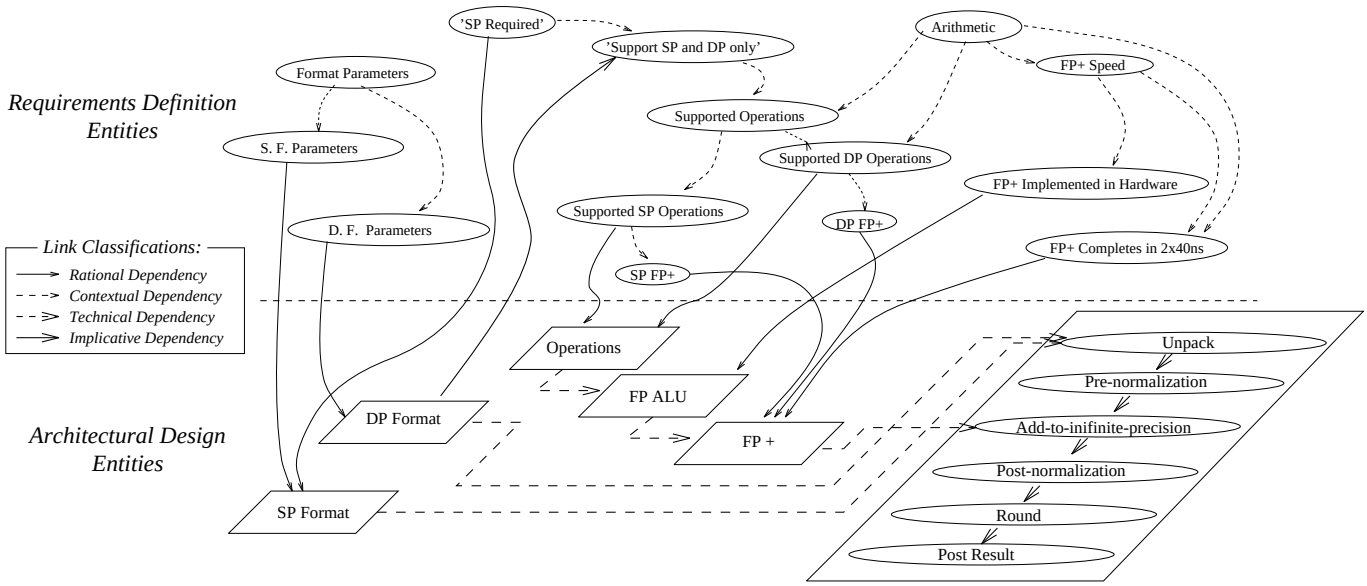

Fig. 4. Linked requirements and design entities

DP_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.

IV. Automated Functional Evaluation

Functional evaluation testing answers the question, "does this 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 [13] are most appropriate.

We implement black-box functional evaluation testing by employing boundary-value analysis and equivalence partitioning techniques [13]. While this is not the only test-case design strategy available, it shows promise of being able to uncover most errors at a reasonable cost, where cost is the number of tests run per error discovered. In general, this type of black-box testing involves the generation of a black-box *testset*, the generation of correct results, the generation of the implemented system's results, and a comparison of these two results.

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 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 5 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**).

The VHDL design entities are directly simulateable, so 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 test. 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 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. These combinations are then checked for their con-

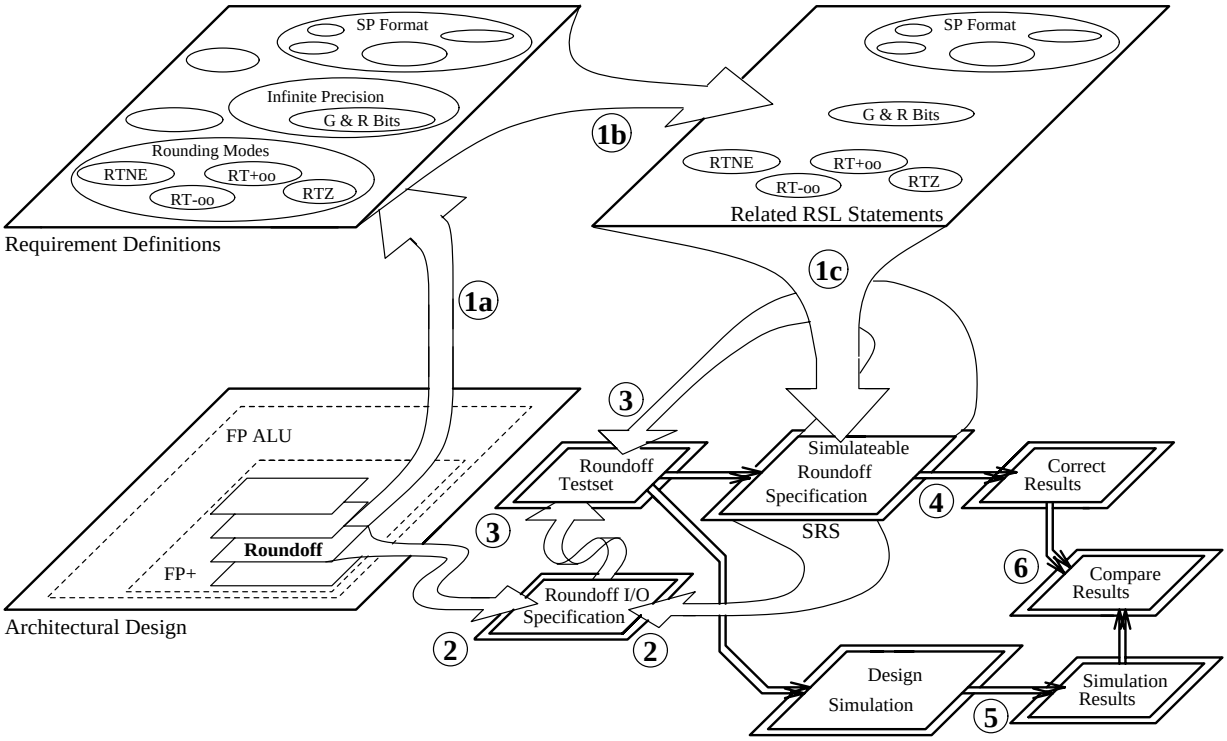

Fig. 5. Black-box functional evaluation for FP ALU roundoff

trol content and for redundancies in order to keep the generated testset from growing unnecessarily large. Other heuristics could be applied as well.

The next step is to apply the testset to the SRS to generate the correct results (**4**). 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.

The fifth step is to apply the testset to the implemented design to generate the simulation results (**5**). Figure 6 shows the evolution from a VHDL design representation to a completed simulation.

In the FP ALU roundoff example, the design representation type is matched with the rules for simulating the representation. In this case there are two means of generating VHDL simulators: one for process (behavioral), and one for non-process VHDL entities. Once the simulation is set up, the testset, formatted for the particular simulator (*testset.vsim_in*) is given as the input to the simulation, and the results are collected for comparison to the 'correct' roundoff SRS simulation results.

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 test setups made available to the designer. We do not address the interface issues, as our emphasis is on generating the information rather than presentation.

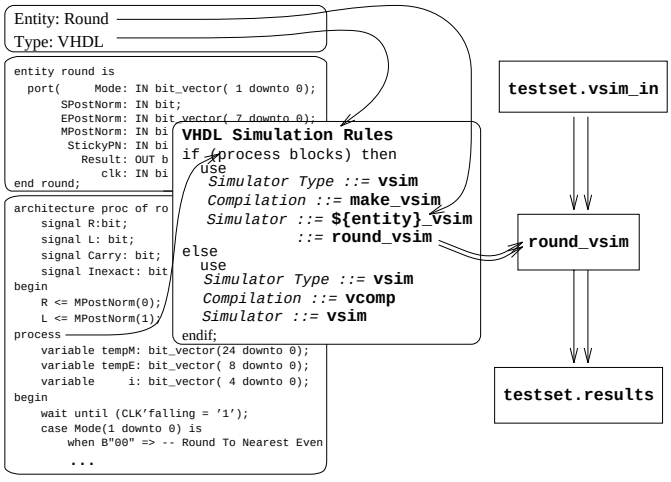

Fig. 6. Simulation of implemented roundoff design

V. Roundoff Example

Consider one small but important aspect of the design of the FP ALU in the DLX, the implementation of roundoff. Our goal is to evaluate the function of this isolated part of the design, and compare it to its requirements. To this end, we 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, using the RD link information, 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 yielding 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 I 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.

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)]	11111111111111111111, 11111111111111111110, 01111111111111111111, 00000000000000000001, 00000000000000000000	5	Max Max - 1 Mid Min + 1 Min
Sticky [StickyPN]	0, 1	2	

TABLE I

REDUCED INPUT TEST CLASSES AND VALUES FOR FP ROUNDOFF

Figure 7 presents parts of the test results for the VHDL design entity *round*, specifically a piece of the tests that correspond to 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.

```
|================================================
| 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
```

Fig. 7. Summary black-box test results for FP ALU roundoff showing two detected errors

The summary results for case 19 of Figure 7 show discrepancies between the correct results (`correct round = ...`) and the implemented results (`round = ...`). As it turns out, this error was caused by some extraneous code in the behavioral design for *roundoff*, which was determined to be the cause of five other errors detected in 800 tests (test cases 20, 59, 60, 159, and 160). Once the extraneous code was removed, subsequent use of the testset discovered no more errors.

VI. SUMMARY

In this paper, we have presented an information process model for design, and used this model as a basis for a unified semantic graph data model for representing linked requirements and design data. Based on this data model, we presented a methodology for automating functional evaluation testing 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. Our unified database is being developed using *ODE* [3], and its O++ language, which is a persistent superset of C++.

REFERENCES

[1] Alford, M. Software requirements engineering methodology (SREM) at the age of eleven: Requirements driven design. *Modern Software Engineering: Foundations and Current Perspectives*, ch. 11, pp. 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] Dar, S., Gehani, N., and Jagadish, H. ODE object database and environment. AT&T Bell Laboratories, CTR Lab, 1991.

[4] 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, Mar. 1992.

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

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

[7] Jackson, M. and Zave, P. Domain descriptions. *Proc. of the IEEE Int. Symp. on Requirements Engineering*, pp. 56–64, Jan. 1993.

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

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

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

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

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

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

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

[15] Smithers, T. Design as exploration: Puzzle-making and puzzle-solving. *Workshop on Search-Based and Exploration-Based Models of Design Process*, pp. 1–21, June 1992.

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