Fault Detection and Localization

Kristine J. Parrella
Andrew Wilmot
Intermetrics, Inc.
7918 Jones Branch Drive, Suite 710
McLean, VA 22102
(703) 827-2606

Abstract

During the development of the Waveform and Vector Exchange Specification (WAVES), the additional need for an exchange format for fault information was recognized. As a result, IEEE PAR 1029.2, Fault Dictionary Language (FDL) was born. This paper discusses the development that has been performed to this point on FDL, including the requirements and guidelines that will shape the development of the standard. These requirements and guidelines have been formulated during the IEEE DASS Test Analysis and Standardization Group working meetings. Usage scenarios for the standard are presented to further demonstrate the intent of the standard. In addition, the organization of the data to be represented in FDL will be examined, and the impact of the "lessons learned" during the WAVES implementation process on the FDL implementation will be discussed. Finally, possible implementations of FDL based on VHDL will be presented.

1. Requirements and Guidelines for FDL

The design of the FDL standard is governed by both requirements and guidelines. The requirements state that the standard shall be capable of representing fault dictionary data required for validation, acceptance, and diagnostic testing. The standard is also required to represent fault information at all levels of an electronic system. Another requirement is that the standard shall provide links to standards that describe the topology of the unit under test, and support techniques that provide fault isolation within fault equivalence classes (i.e. guided probe). The standard shall also represent the fault information used to characterize the fault identification (e.g. fault type, fault location), the fault information used to characterize the diagnostic capability of a test (e.g. fault coverage) and any necessary configuration management information.

The guidelines governing the design of the FDL standard suggest that the standard should be capable of representing fault dictionary data resulting from fault simulation. In addition, the standard should represent accessibility information on signals and components in some manner, as well as minimizing the amount of data redundancy. The standard should not describe information that may be expressed using WAVES, VHDL, or the Test Requirements Specification Language (TRSL IEEE PAR 1029.3). Any FDL package specifications that are represented in VHDL should conform to the semantics of VHDL, and the standard should also be based on the same VHDL constructs and package specifications as WAVES. The standard should be extensible to support new fault dictionary techniques in the future.

The above only summarize the requirements and guidelines governing the design of the FDL standard. To obtain the latest and most accurate version of the requirements, consult the requirements document listed in the references section of this paper.

2. Usage Scenarios for FDL

Before examining the organization and implementation of FDL, it would be beneficial to introduce the anticipated usages of the standard. The primary usage is the exchange of the necessary fault dictionary information between design and test. In addition, the standard should facilitate the porting of fault dictionary data from one test environment to another. Once the data has been ported into a test environment, it should be possible to use it, where necessary, to generate a new test program.

There are also several possible scenarios in which the manipulation of the data represented in FDL would be required. For instance, metrics calculations might be performed on the data. Specifically, the fault coverage for a test might be calculated. The fault dictionary might be searched to isolate possible causes of failures, or the data contained in the fault dictionary might be used to aid in the performance of guided probe. The user might also want to extract data for a testability analysis of their design

The FDL data might possibly be used in a simulation environment. For example, information might be extracted from FDL about how the outputs of a unit under test (UUT) would behave when a certain fault has occurred in the UUT. These faulty outputs might then be used in a simulation of the system that used the faulted unit, to see how the system would respond to the faulty behavior.

3. Data Organization

Given the existing fault dictionary practices, an attempt has been made to decompose the data which must be represented in FDL into primitive information elements. Relationships have been defined among these primitives, in conjunction with operators on both the primitives and the collections of primitives.

Before describing the organization of the primitive elements, it would be useful to discuss the dependency of FDL on information represented in other standards. Figure 1 illustrates these relationships. As seen in the illustration, the fault dictionary is dependent on the test data, as well as the product data. The test data will not be represented in FDL. Instead, the fault dictionary will refer to the appropriate test vectors, test pins, etc. In the same manner, the fault dictionary will make the necessary references to the product data. This will ensure that there is no duplication of information in FDL.

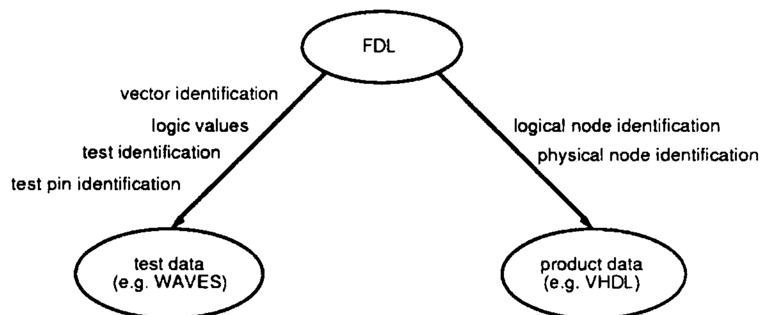

Figure 1
Relationship Between FDL and Other Standards

The organization of the primitive elements within the fault dictionary can now be examined. The information models which document the fault universe and fault dictionary are represented using Express-G.

Figure 2 shows the model for a fault set. In the context of FDL, a fault set is simply a set of zero or more unique faults, where a fault is defined as a circuit defect which causes the circuit to malfunction. The fault universe is an instance of a fault set which includes all known possible faults for a given circuit. Certain identification information must be provided for each fault, as illustrated in Figure 2. This includes information about the logical location of the fault in the UUT, as well as the fault type and the fault origin. In this model, the fault type is an enumerated type. Example values of fault type might be 'stuck at high', or 'open floating low'. The fault origin is also an enumerated type which documents the basis for the inclusion of the fault in the fault set. Example values of the fault origin might be 'predicted by simulator', or 'learned by engineer during debug phase'.

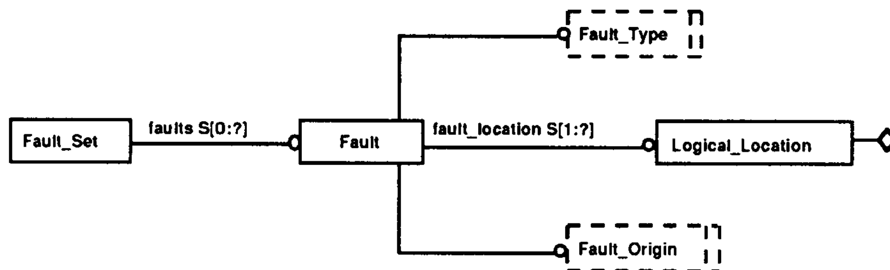

Figure 2
Fault Set

The fault dictionary contains data on the response of the faulted UUT to a test. During a test, each fault will cause the resulting primary output pin values to differ from the expected primary output pins values, if the fault is detectable. The fault dictionary correlates the failure modes and the faults that may have caused the failures.

Certain information must be provided for each UUT failure mode, as illustrated in Figure 3. This includes the identification of the test vectors that will fail, the signature of the failing outputs, and the probability that the test vector will fail. The ambiguity group must also be provided for each bad UUT response. The ambiguity group is a set of faults. If any of these faults were to occur in the UUT, the UUT would fail in the manner described The set is named the ambiguity group because, given the bad UUT response, there is no way to immediately tell which fault in the ambiguity group caused the failure.

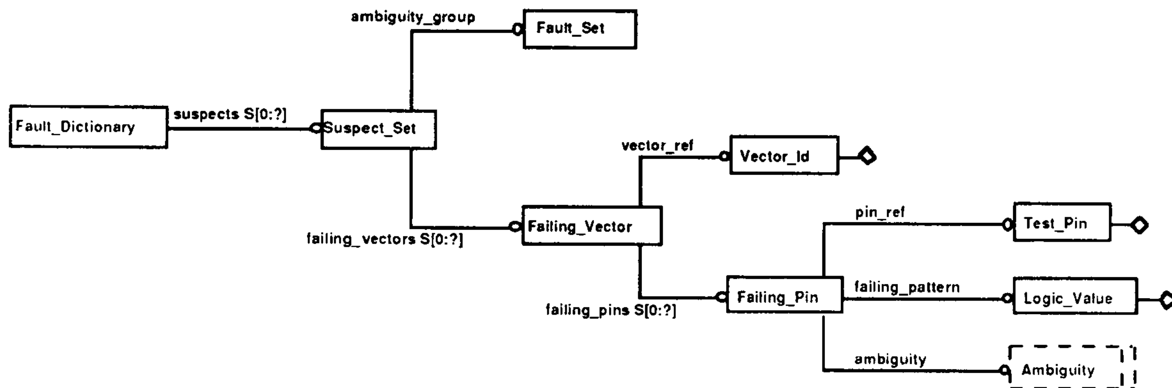

Figure 3
Fault Dictionary

As an example, consider a UUT with outputs that differ from the expected outputs at test vectors 13, 47, and 48. The information that would be represented in the fault dictionary is shown in Figure 4. Each failing vector would be identified. In this example, the vector number provides this identification. There would also be information about each primary output that failed at that vector. At test vector 13, primary outputs Q0 and Q3 exhibit failures. Primary outputs Q0 and Q3 both fail low at this test vector. There is also information about the ambiguity associated with the failure. For instance, the failure on Q0 at vector 13 is N/A, or non-ambiguous. The failure will always be apparent on the output Q0. However, the failure on Q3 at vector 13 is A, or ambiguous. The failure may only be apparent on the output Q3 under certain conditions.

Based on the failing response of the UUT to these test vectors, an ambiguity group can be identified. The ambiguity group would be the set of faults that may cause the UUT to fail in the manner described by the failing vector information. In this example, there are two faults that may have caused the UUT to fail test vectors 13, 47 and 48. However, since both the faults in this ambiguity group would cause the same test vectors to fail, there is no way to select the specific fault responsible for the failure without further isolation.

Ambiguity Group

(U1.Q, stuck_at_0)
(U27.A6, stuck_at_1)

Failing_Vector

	Primary Output			
Vector_Id	Q0	Q1	Q2	Q3
13	L N/A			L A
47		L N/A		
48		H N/A	H A	

Figure 4
Fault Dictionary Suspect Set Example

Abstract operators may be defined on data representing fault set and fault dictionary objects. Although the names of the operations differ, they all basically have the same functionality. One operation would be the formation of an object from a set of heterogeneous information. For instance, given a fault type, a logical location, and a fault origin, the user must be able to build a fault. Conversely, given an object, the user must be able to extract any piece of information that was used to build the object. The user must be able to add an object to a set of objects. An example of this would be the addition of a new fault to the fault set. The user must also be able to merge two sets of object or remove an object from a set. In general, support must be provided for all set operations.

4. Implementation in VHDL - Ideas from WAVES

The requirements document suggests that FDL should be based on VHDL. Before addressing the actual implementation of FDL, it is helpful to analyze the WAVES development, along with some of the "lessons learned" during that process. These ideas may then guide FDL development.There are several principles which affected the development of WAVES which can be considered. These principles include inter-operability, information hiding, data hierarchy, and extensibility.

Inter-operability with existing standards is an important consideration. Why should FDL be based on VHDL and WAVES? The primary reason is these are leading standards for the design/test data (see Figure 1) needed for fault dictionary implementation. Also, the basic language syntax, concepts and definitions found in VHDL and WAVES can be re-used in FDL. Another point is that the user is potentially familiar with the construct

name references in VHDL and WAVES. If FDL is based on VHDL and WAVES, it should be fairly easy for the user who knows VHDL to pick up the new standard and comprehend the information that is being conveyed. If required, VHDL models may make use of information specified by FDL. From another point of view, it should also be easy for a tool producer to base new FDL tools on their existing VHDL/WAVES tools. The tool producer's existing "design database" technology can be used to also store fault information.

Information hiding is also a desirable implementation feature. If the implementation decisions about important data types can be kept hidden from the user, the implementor will be allowed greater flexibility. The user only needs to be presented with the name of the data type, and the specifications of operations that they can use to manipulate that data type. The WAVES approach to information hiding in a VHDL implementation provides the user with package(s) of data type declarations, and operations on the data types. This combined with semantic restrictions on the (subset) language allows the implementation of "private" types in VHDL. The data is accessible only by the allowed declared operations.

If the data is stored in a hierarchical manner, the benefits include ease of understanding, and ease of construction. Furthermore, data compaction is possible in the implementation. Data which has been previously described at a different level of the hierarchy can be referenced, rather than respecified. Since one of the FDL guidelines is to compact the data wherever possible, this bears directly on the FDL implementation approach.

Another guideline for FDL is that the standard should be easily extensible to support new fault dictionary practices. There are several ways in which this can be achieved. First, the user may be provided with the ability to express some familiar concepts as combinations of more primitive concepts. The user may also be given the ability to form new concepts in this manner. As an example, consider the logic value in WAVES. In the value dictionary, the user builds up the definition of each logic value,as a combination of state, strength, direction and relevance. Although each of these primitive concepts is pre-defined by an enumerated type, it is the user who places them into the combinations which represent the logic values.

Extensibility may be promoted by allowing the user to define key concepts, such as basic types and operations upon those types, in a standard framework. Uninterpreted information could also be conveyed within the standard. This would allow the evolution of "standard practices". Finally, a mechanism could be provided that would allow the standard to be "escaped".

5. FDL Implementation Approaches

A strawman implementation FDL approach has been developed. This implementation is based on VHDL, and attempts to follow the basic principles outlined in the previous section. This section contains a segment of that approach. This segment provides the support for the fault set. In addition, a small example which builds a fault universe using the implementation approach is provided.

The following package specification supports the principle of information hiding. The actual type declarations are hidden from the user, as well as the implementation of the functions that operate on these types. The actual implementation can be written in VHDL, or in another language that will more efficiently manipulate the data.

```
library fdl_std;
package fdl_visible is
```

```
    subtype fault_kind is fdl_std.fdl_hidden.fault_kind_type;
    subtype fault is fdl_std.fdl_hidden.fault_type;
    subtype fault_set is fdl_std.fdl_hidden.fault_set_type;

    function "+" (Fault1, Fault2 : in Fault) return Fault_Set;
    function "+" (Fault1 : in Fault;
                  Set1 : in Fault_Set) return Fault_Set;
    function "+" (Set1 : in Fault_Set;
                  Fault1 : in Fault) return Fault_Set;
    function "+" (Set1, Set2 : in Fault_Set) return Fault_Set;

end fdl_visible;
```

Next follows the source code to build a simple fault universe. Specific pieces of heterogeneous information must be supplied in the source code for each fault. The user supplies a string which names the fault. Next follows the type of the fault, and the logical location of the fault. These faults are then combined to form the fault universe.

```
library fdl_std;
use fdl_std.fdl_visible.all;
package body fdl is

    procedure Build_It is
        variable Universe : Fault_Set;
    begin
        Universe := ("L1@0", stuck_at_0, "U3.Y") +
                    ("L1@1", stuck_at_1, "U3.Y") +
                    ("L2@0", stuck_at_0, "U10.Q") +
                    ("L2@1", stuck_at_1, "U10..Q") ;
    end Build_It;

end fdl;
```

This preliminary segment of code raises many implementation questions. For instance, certain types seem to lend themselves to being user-defined. It might be possible to given the fault_kind type a low level definition, and then have the user map their own high level definition to that low level definition. The way that information contained in VHDL and WAVES source code is referenced must also be defined. This includes the specification of the logical location pathnames in FDL, as well as the identification of instances in procedurally generated sequences.

Due to the large amounts of data being conveyed by FDL, it might be advisable to have the capability to store data in an external file with a fixed format. The FDL standard may then define procedures which would interpret the data in the correct manner. In this same vein, the simplicity of a purely declarative implementation should be compared with the compactness of the proposed implementation which includes procedural capabilities.

6. Summary

Work is on-going on the Fault Dictionary Language standard. This paper has attempted to serve as an introduction to the requirements and guidelines which are influencing the development of the standard. In addition, the organization of the data to be represented in the standard, implementation ideas and issues have been introduced. As development continues on the standard, feedback from the community on the material presented here would be welcomed.

7. Acknowledgments

This work was funded in part by the Air Force's Rome Laboratory, Griffiss Air Force Base; under contract F30602-90-D-0097.

8. References

IEEE Standard VHDL Language Reference Manual. IEEE Standard 1076-1987.
IEEE WAVES working documents, available from committee.
IEEE FDL working documents, available from committee.