5 Netlist Comparison

In netlist comparison, we compare two netlists to verify their supposed equivalence. The two netlists are usually from different sources (such as an S-Edit schematic and an L-Edit layout), but they can also be from two schematics

or two layouts; generated by different design editors but represented in the same netlist format; or two revisions of the same schematic.

In a verification run, LVS first reads in the two netlists and compiles a list of elements and a list of nodes. It then uses an iterative process to repeatedly divide classes of elements and nodes into smaller and smaller classes until each element and node can be uniquely identified and compared.

Netlist Comparison Basics

LVS compares elements and connections for similar characteristics, not for functionality or purpose. Therefore, it is important that the netlists being compared have the same types of basic circuit elements. If you compared a netlist containing Boolean logic gates and a netlist containing transistors, for example, LVS would find them unequal because it does not construct Boolean logic gates from transistors.

LVS processes subcircuits by flattening them, then individually comparing their constituent elements and nodes. If the two designs resolve to the same hierarchical levels, however, you can explicitly define higher-level subcircuits as elements for comparison in an element description file. For further information on this technique, see Element Description File Example on page 3-237.

LVS begins by reading in the two netlists and compiling a list of elements and a list of nodes. (An *element* is any type of logic or circuit component, such as a transistor, resistor, or capacitor. A *node* indicates a connection: a wire and anything directly attached to it.)

LVS then sorts the elements and nodes into classes. A *class* is a set of elements or nodes with something in common. For example, LVS might begin by separating the elements into different classes: a transistor class, a resistor class, and so on. Further divisions might divide the transistor class into P transistor and N transistor classes.

Nodes are separated in a similar manner. They might be separated into classes according to the number of elements attached to them. Further separation might be based on the types of elements attached to the nodes.

The process of separating elements and nodes according to topological information is called *topological matching*. Topological matching groups elements and nodes by types and connectivities, rather than by capacitance or element size, which are not used in the default matching process. Topological matching continues until no further fracturing is possible. Ideally, each class will contain only two members at this point, one from each netlist file. If this state is achieved, the netlists are said to be topologically equal.

If topological equality is not achieved, there can be several explanations. For further information on resolving topological inequality, please see:

- Fragmented Classes on page 3-199
- Automorph Classes on page 3-202
- Permuted Classes in Digital Designs on page 3-207

Fragmented Classes

After topological matching, any class remaining with a different number of members between the two files is called a *fragmented class*. Fragmented classes are almost always the result of differences between the two netlists, indicating a design error that must be resolved.

LVS cannot resolve fragmented classes because there is not a one-to-one match between elements or nodes in the netlists. If a fragmented class occurs, you should examine the source of the netlist files to determine and resolve the problem.

Resolving Fragmented Classes

When iteration produces fragmented classes, you must examine the fragmented class members and trace them back to their origins in the netlist sources.

Understanding LVS output can also help you locate an element or node. During the comparison, hierarchy information is appended onto elements and nodes. For example, an element named **M7(X3/X2)** refers to a transistor element named **M7**, which resides in the subcircuit **X2**, which is instanced from the subcircuit **X3**. See the LVS Output Tutorial on page 3-214 for more information about the LVS output format.

If you cannot identify an element or node from its LVS-generated name, try locating the element or node in the netlist file, which may contain additional information. For example, a netlist generated with the option **Write device coordinates** (in **Tools > Extract—Output**) might contain a line such as:

```
R0 1266 1269 259.6
* R0 Plus Minus ( L B R T ) A = 9.744e2, w= 2.4
```

The letters ***L***, ***B***, ***R***, and ***T*** in the comment line represent four numbers in the netlist file. These numbers would indicate the left, bottom, right and top boundaries of the element recognition layer, respectively. You can look at these coordinates on your layout to find a specific element.

You can also use the option **Label all devices** (in **Tools > Extract—Output**) to label all the devices on the layout with ports, where the port text is the device name. Using **Edit > Find**, you would then be able to find a specific device in the layout.

If the fragmented class is a node, you can count the number of pins on the nodes. This will tell you how many elements are attached to the node.

Another way to identify fragmentation problems is to compare the fragmented classes. If one netlist produces two fragmented element classes with a fanout of one, and the other netlist file produces one fragmented element class with a fanout of two, these three classes may represent the same element (with one pin left without a connection). In such a case, LVS can identify floating pins if you

select the option **Detailed processing information** in **Setup Window—Verbosity Level** (page 3-169).

Automorph Classes

After topological matching, any unresolved class containing an even number of members, half from one netlist and half from the other, is called an *automorph class*. Automorph classes are not necessarily caused by design errors—they also occur when LVS does not have enough information to distinguish between members of a class. You should resolve automorph classes, however, to confirm the correspondence of the two netlists.

In some cases, members of an automorph class actually do match each other, such as when a class contains identical elements connected in parallel. Because LVS cannot distinguish such elements using its default iteration procedure, it may be unable to resolve them during its initial verification run. For example, LVS would be unable to distinguish two equivalent resistors connected in parallel. In practice, however, such identification would be unnecessary, because the resistors are identical in all respects.

Resolving Automorph Classes

The following sections describe three methods for resolving automorph classes:

- Preiteration Matching, below
- Detailed Trial Matching, below
- Parameter Matching on page 3-205

Preiteration Matching

In some cases, you can prevent the formation of an automorph class by providing LVS with enough information to distinguish between the members of a class. This technique is called *preiteration matching*, because LVS performs a preliminary match of specified elements or nodes before its usual iteration.

LVS performs preiteration matching when you instruct it to use a *prematch file*, which contains a list of statements that define equivalent elements or nodes. You enter these statements in a prematch file when you know that a given pair of members are, in fact, identical. The exact format of this file is described in Prematch File Format on page 3-266.

To specify the use of a prematch file, check the **Prematch file** option in **Setup Window—File** (page 3-158) and provide the filename and path in the adjacent field.

Note: Preiteration matching can significantly increase verification speed, especially for large designs, but it can also prevent LVS from detecting design errors if the prematch file contains erroneous matches. Use a prematch file only when you have previously verified the existence of an automorph class and only when you are certain that the specified elements or nodes are equivalent.

Detailed Trial Matching

Another means of resolving an automorph class is to instruct LVS to run *detailed trial matching*. In detailed trial matching, LVS attempts to resolve automorph classes by making a "guess" match between two class members, then resuming the iteration from that point. The program begins by matching an automorph element pair and then iterates on the automorph node classes until no more iteration can be done. It then matches a pair of automorph nodes and iterates on the remaining automorph elements. LVS continues in this way, alternating between element and node classes until it can go no further.

Detailed trial matching sometimes fractures an automorph class into a fragmented class, but this result would indicate that LVS made an incorrect trial match. In such a case, you should try one or more of the following solutions to resolve the automorph class:

- Rerun the verification with detailed trial matching enabled. LVS temporarily stores a record of its matches in memory, but it does not save them with the VDB file. Therefore, you must make the second attempt at detailed trial matching without closing the VDB file.
- Examine the output file to discover what matching assignments LVS made. With your knowledge of the design, you may be able to make better assignments than those made by LVS and enter them into a prematch file.
- Rerun the verification with parameter matching enabled—see Parameter Matching, below.

To specify detailed trial matching, check the option **Detailed trial matching to resolve automorph classes** in **Setup Window—Performance** (page 3-167). In the event of an automorph class, LVS will automatically proceed to detailed trial matching. If you do not check this option before the verification run, LVS will prompt you for permission to perform detailed trial matching.

If you run the iteration in batch mode or as part of a verification queue, LVS automatically performs detailed trial matching on all automorph classes whether you check the option or not.

Note: LVS stores element and node information internally, and the program's matching assignments for a particular automorph class depend on the size and location of memory blocks available at a particular time. Therefore, results from detailed trial matching may vary with each run.

Parameter Matching

The third method of resolving an automorph class is called *parameter matching*. In this method, LVS considers additional user-specified parameters such as capacitance and element size to further distinguish class members that would otherwise form an automorph class. For example, it may be possible to divert automorph nodes with different capacitances into different classes, possibly resolving the automorphism. Such a step may also convert an automorph class to

a fragmented one, but you can then take appropriate steps to correct the error that produces the fragmented class.

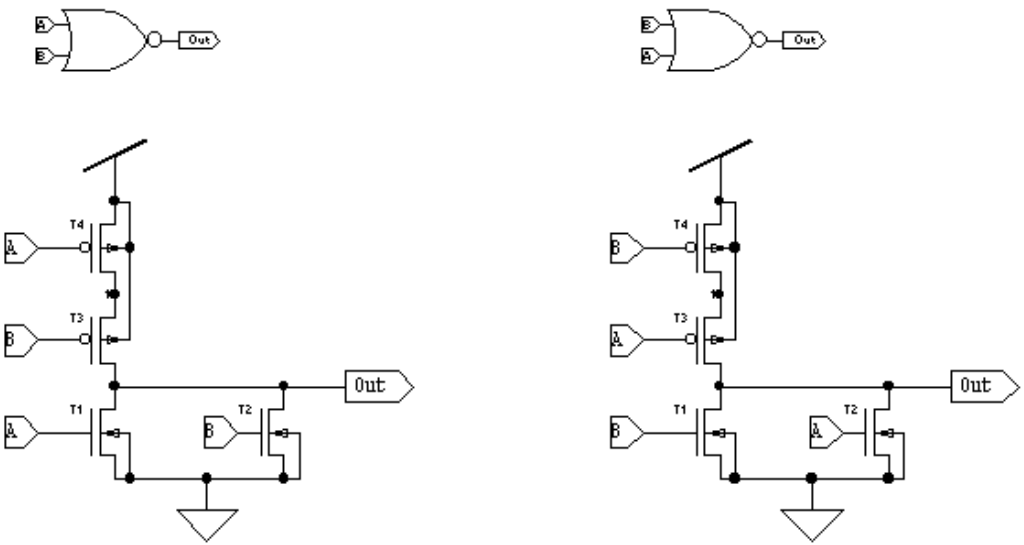
To specify parameter matching, select the particular parameters you want LVS to consider in the dialog **Setup—Element Parameters**.

Permuted Classes in Digital Designs

If you select the option **Replace series chain MOSFETs**, LVS can catch and identify permuted classes. A *permuted class* occurs in digital designs when many digital circuits provide terminals that are functionally equivalent, but in a different order in schematic and layout designs. In such a case, LVS will create fragmented classes unless it considers permuted classes.

Warning: Permuted classes in an analog design could generate problems and should be avoided.

The following illustration demonstrates a permuted class. The two input NOR gate designs shown below are functionally identical, but in topological iteration with series replacement, LVS would note the pin permutation and report an error. The iteration invoked without series replacement would generate four fragmented element classes, each with one element in the class.



Avoiding Permuted Classes

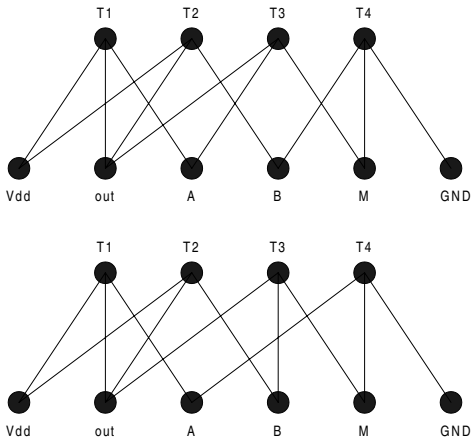
The best way to avoid permuted classes is to standardize pin name assignments in both schematic and layout design. Always assign **A** to the top pin or the pin closest to **Vdd**, for example, and **B** to the lower pin or the pin closest to **GND**.

Standardized pin name assignments such as these will prevent LVS from generating permuted classes.

LVS Algorithms and Limitations

LVS uses an algorithm that repeatedly fractures the classes in an effort to generate a unique classification of the elements and nodes in a netlist. This methodology has several important advantages, but it also has a few limitations.

During the topological matching process, LVS continues to fragment elements and nodes into increasingly smaller and more identifiable classes. In an ideal comparison, the iteration process ends when each class has exactly two members whose features are known. The following diagrams illustrate this procedure.



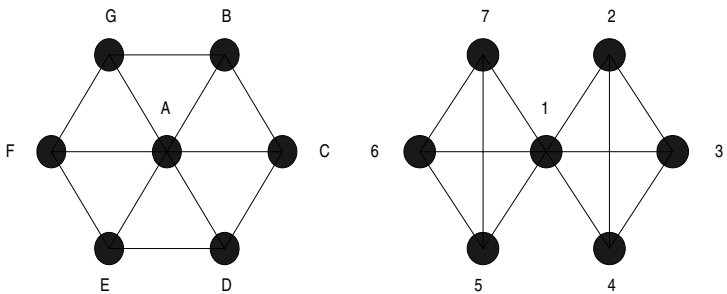
Notice that bulk nodes are not considered, and the two diagrams of the internal data structures are not identical, indicating fragmented classes if LVS is run without series replacement.

There are several advantages to representing netlists in this way, including:

- Isomorphism can easily be determined since each representation is unique for each circuit.

- No knowledge of driving elements is required.
- No knowledge of the elements and their pins is required.
- There is easy recognition of shorts and opens.

One limitation to this type of netlist representation is that a highly symmetrical design will fail to converge to fragmented classes, as in the diagram below. Exhaustive trial matching of all elements in the second equivalent class yields no self-consistent partitions, and the data structures are not isomorphic. The result, however, is correct.



Resolving Discrepancies

Resolving netlist comparison discrepancies can be a challenging process if the involved circuits are large, because a single problem might create many automorph and/or fragmented classes. Here are a few suggestions for resolving netlist comparison problems, which are applicable to the resolution of both automorph and fragmented classes.

- Liberally label the sources for your netlists. If you do not provide a name for a node or element, LVS constructs one automatically, but the resulting name may be just a number, perhaps concatenated onto another string. Even an LVS-generated identifier can be helpful, however—if you recognize only a portion of a node or element name, you may still be able to identify the problem.
- If possible, recompare the netlists often. Sometimes identifying a single matching element or node will provide LVS with enough information to complete the job. In other cases, a single problem will create a very large fragmented class. Identifying one or two matching pairs from this class and entering them into a prematch file will usually be sufficient for LVS to fracture the large fragmented class into several smaller ones, thus giving you an easier task to perform.