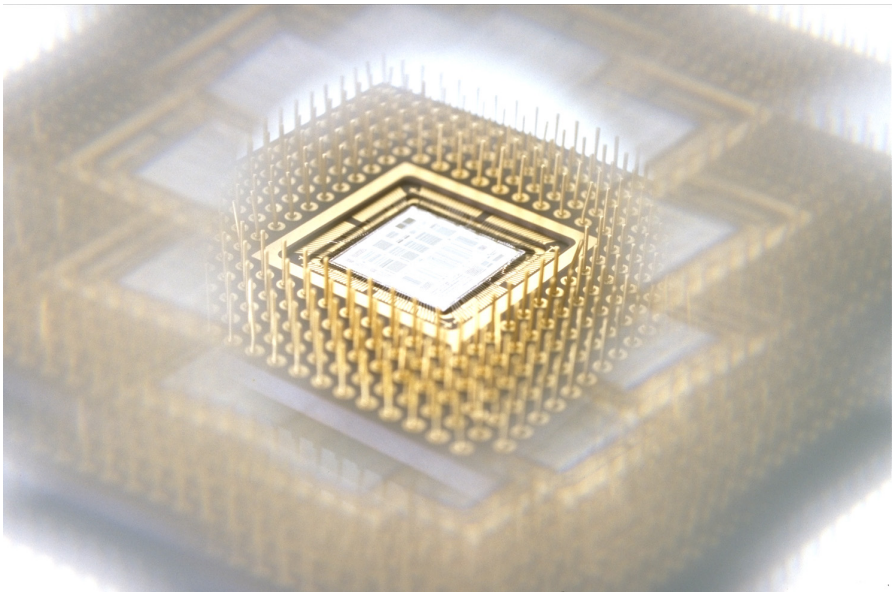
Avertec Tools

GNS
User Guide



Software Release 3.4p5

June 7th, 2010

About this Document

This document explains:

- The input formats supported
- How to perform the hierarchical recognition
- User constraints
- How to use GNS

Documentation issued and compliant with Avertec Tools Release 3.4p5.

Please contact support@avertec.com for comments relating to this manual.

Table of Contents

Chapter 1. Hierarchical recognition with GNS

1.1. Overview

The use of automatic tools in the design of integrated circuits allows the generation of complex circuits made up of a large number of blocks of identical structure. In many cases these blocks represent a significant part of the circuit, and require a particular attention of the designer during verification in order to validate the structure generated by the automatic design tools.

The Yagle tool automatically generates a functional description without requiring any a priori knowledge of the circuit. This capability is ideal in the case of digital circuits made up of many diversified structures. However, rule-based recognition, the alternative to the automatic approach, is significantly more efficient in the handling of repetitive structures. Furthermore, pattern matching is the only way to cope with mixed analog / digital designs.

Embedded memory cores are the perfect example of the use of regular, repetitive structures. A large part of the design is made up of the arrays of memory cells together with analog blocks for pre-charging and amplification. Such designs can be represented by highly compact, hierarchical structural descriptions.

GNS allows the user to provide hierarchical abstraction rules in a similar way to the original structural description.

Hierarchical recognition of regular structure blocks from transistor level allow to perform faster validation on high complexity circuits. Performing the validation at transistor level guarantees the precision of the validation, since the transistor level is the most accurate description of the final circuit.

GNS provides the means for today's circuit designers to satisfy the demand for rapid time-to-market whilst enhancing the robustness of their validation strategy.

The most important application of the GNS tool are:

- Combined structural and functional verification of custom blocks.
- Automatic functional characterization of embedded memory cores.
- High speed formal verification of extremely large, regular designs, such as memory cores.
- Verification of in-house structural design rules.

1.2. Description

The GNS tool offers designers the possibility of writing generic rules to verify the structural integrity of all or part of their circuits. Each recognition rule is written in a syntax compatible with VHDL. The rules are fully hierarchical, allowing concise descriptions of complex structures. In addition, each rule can be made generic, meaning that interconnections of arbitrary numbers of components can be represented.

- The input file is a netlist of transistors (including capacitances and resistances of the interconnecting wires) extracted from the layout in SPICE format.
- GNS Isolates all circuit structures matching the user-defined recognition rules.
- Each recognition rule can be associated with an ACTION which is executed upon validation of the rule.
- ACTIONs can be used for example to customize VHDL or Verilog code generation for particular structures.

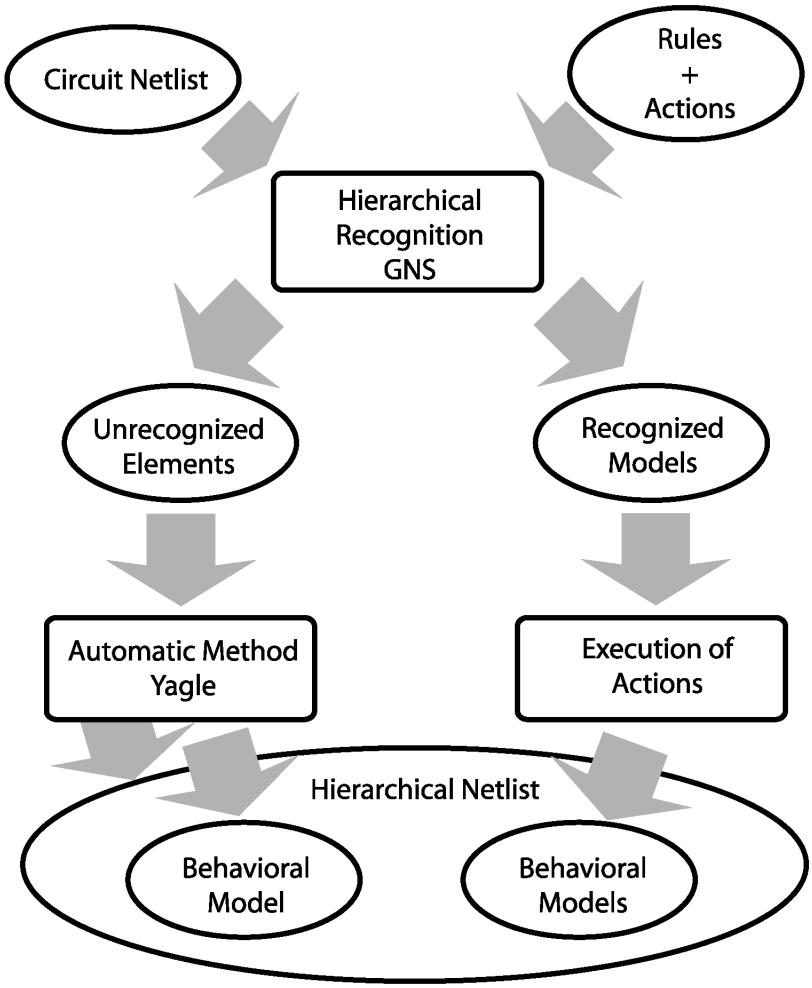
1.3. Hierarchical Generic Recognition

GNS combines functional and structural verification in a single tool. Structural verification is performed by the application of user-defined hierarchical rules, starting at the transistor-level. These rules, written in VHDL, specify interconnections of a fixed or arbitrary number of components. The structure of even complex circuit architectures can be fully described.

Behavioral models can be automatically generated even for totally generic structures. This is done by associating actions with the recognition rules. Each action is a small program, written by the user, in a subset of 'C' which dynamically generates the model based upon the sizes of the recognized structures.

1.4. Integration with Yagle

With the addition of the GNS module, Yagle is a complete functional abstraction solution. This technology, coupled with the FCL technology to isolate basic building-blocks, allows rapid verification of the regular structures in a design using compact user-defined rules.



The circuit is automatically partitioned into regular and non-regular parts. Yagle calculates a functional model which seamlessly integrates with the models generated by GNS to allow simulation and verification of the complete design.

Chapter 2. Configuration Variables

2.1. License Server

avtLicenseServer
`<string>` Hostname of the machine running the license server

avtLicenseProject
`<string>` Project name. Used in license logging.

2.2. Environment

avtLibraryDirs
`<string>` The set of library directories which are scanned for required subcircuits.

avtBlackboxFile
`<string>` Name of the file containing the cells to exclude of analysis.

avtCatalogueName
`<string>` File containing a list of subcircuits to be considered as leaf cells when flattening a design. Each line in this file refers to a single subcircuit, with the format `<subcircuit> C`. The default value is `CATAL`.

2.3. Names

avtVddName
`<string>` Name of any signal or connector which is to be considered as power supply (a `*` in the name matches any string). Several names, separated by `:`, may be specified.

avtVssName
`<string>` Name of any signal or connector which is to be considered as ground (a `*` in the name matches any string). Several names, separated by `:`, may be specified.

avtGlobalVddName

`<string>`	Name of an internal signal to be considered as power supply (a `*` in the name matches any string). Signals in different subcircuits of a hierarchical netlist with a name given here will be considered as equipotential and this name will be used in the flattened netlist. This is identical to the use of the .GLOBAL directive in a spice netlist. Several names, separated by `:`, may be specified.

avtGlobalVssName

`<string>`	Name of an internal signal to be considered as ground (a `*` in the name matches any string). Signals in different subcircuits of a hierarchical netlist with a name given here will be considered as equipotential and this name will be used in the flattened netlist. This is identical to the use of the .GLOBAL directive in a spice netlist. Several names, separated by `:`, may be specified.

avtCaseSensitive

`yes`	Upper and lower case characters are distinct
`no`	Upper and lower case characters are seen as identical
`preserve`	Default, upper and lower case characters are seen as identical but the original case is preserved

avtInstanceSeparator

`<char>`	Character used to separate instance names in a hierarchical description. Default value is `.`

avtFlattenKeepsAllSignalNames

`yes`	When flattening a netlist, each signal keeps all its names through the hierarchy.
`no`	Default, only one name (the shortest) is kept per signal.

avtVectorize

Controls the internal representation of vector-signals.

`yes` Default, vector-signals are represented internally as vectors, as far as the vector indexation is one of [], <>, _. For example, if both `foo[1]`, `foo<1>` and `foo_1` appear in the source file, they will all be represented internally as `foo 1`

`no` Vector signals are represented internally as they appear in the source file.

`<string>` Explicits the vector-signals indexations that will be interpreted as vectors, and the represented internally as vectors. `string` is a comma-separated list of single or paired delimiters. For example, if `string` is set to `"[],_"`, only `foo[1]` and `foo_1` will be represented internally as `foo 1`.

Special attention should be paid to the Verilog case. Verilog only accepts `[]` as legal vector indexation. Legal verilog vectors are represented internally as vectors if `avtVectorize` is different to `no`.

Illegal Verilog vectors are supported and controlled by `avtVectorize` as far as they are escaped and `avtStructuralVerilogVectors` is set to `yes`. For example, `\foo<1>` is represented internally as a vector if `avtStructuralVerilogVectors` is set to `yes` and `avtVectorize` is set to `<>`.

2.4. Pattern Matching

fclLibraryName

`<string>` Name of the file containing the list of cells in the user-defined cell library used. The default is `LIBRARY`.

fclLibraryDir

`<string>` Access path to the directory containing the user-defined cell library used. Default is a directory `/cells` in `avtWorkDir`.

fclGenericNMOS

`<string>` A colon separated list of transistor model names which the FCL pattern-matching engine considers will match to any N-type transistor. If a pattern netlist contains non-generic N-channel transistors then these transistors will only match to transistors with an identical model. Default is `tn:TN`.

fclGenericPMOS

`<string>` A colon separated list of transistor model names which the FCL pattern-matching engine considers will match to any PMOS transistor. If a pattern netlist contains non-generic P-channel transistors then these transistors will only match to transistors with an identical model. Default is `tp:TP`.

fclWriteReport

`yes` A correspondence file is created if the -fcl option is used. This file details all the recognized instances.

`no` Default

fclAllowSharing

`yes` Matched cells are allowed to share transistors.

`no` Default

fclCutMatchedTransistors

`yes` Matched transistors are eliminated from the transistor netlist. Results in a strict partitioning of the cones and the matched cells.

`no` Default

fclMatchSizeTolerance

`<int>` Percentage tolerance for matching transistor sizes.

fclTraceLevel

`<int>` Number greater than 0. Trace information is displayed during the pattern-matching phase.

fclDebugMode

`<int>` Number greater than 0. Additional debugging information is displayed during the pattern-matching phase.

2.5. Hierarchical Pattern Matching

gnsLibraryName

`<string>` Name of the file (recognition library) containing the list of cells to recognize. Default is `LIBRARY`.

gnsLibraryDir

`<string>` Access path to the directory containing the recognition library. Default is directory `cells/` in `avtWorkDir`.

gnsKeepAllCells

`yes` All matched structures are extracted from the netlist.

`no` Default

gnsTemplateDir

`<string>` Directory where to find the GNS templates. Default is `$AVT_TOOLS_DIR/gns_templates`.

gnsTraceLevel

`<int>` From 0 to 6. Indicates the level of trace displayed during the recognition phase. Default is 0.

gnsTraceFile

`<string>` Name of the output trace file. Default is stdout.

gnsTraceModel

`<string>` When tracing the recognition, indicates the name of the recognized model to trace. If not specified, traces all models.

gnsFlags

This configuration controls the behavior of GNS. The values (flags) are added separated with commas. Available flags are:

`EnableCore` Enable the generation of a core file for a crash in a user compiled API.

`NoGns` Disables the generation of the `.gns` file

`VerboseGns` Produces a more readable `.gns` file.

`NoOrdering` Disables the top-level instance connectors reordering. Should not be set if using the BEG functions.

2.6. API Specific

apiFlags

Controls the behavior of the GNS API. The values (flags) are added separated with commas. Available flags are:

`ttvUseInstanceMode`	Sets the TTV functions to generate/use one timing view per instance of the same matched subcircuit.
`ttvDriveDTX`	Enables the drive of the `.dtx` and `.stm` files for timing views created with the TTV functions

Chapter 3. Performing the Hierarchical Recognition

3.1. Description

GNS is a hierarchical recognition tool for digital and mixed-signal circuits. Starting from a flat transistor level circuit description, the tool identifies the hierarchical structure within the circuit based on user-defined generic recognition rules. The recognition rules are written in a subset of structural VHDL. The genericity of the rules comes from the use of generics in VHDL. These generics are used for loop variables of VHDL GENERATE statements. In standard VHDL, the value associated with the generics is assigned by the higher level instantiating description, but in GNS the values are assigned according to what is identified in the circuit. Whenever a recognized structure is important for the user, the tool can be programmed to generate automatically, through ACTIONs, a behavioral description of the entire structure using the computed generic variables.

In the first phase, GNS transparently uses the FCL pattern-matching technology integrated in Yagle to identify transistor level blocks. In the second phase, GNS tries to apply the hierarchical rules from the lowest level of the hierarchy composed of the recognized transistor level blocks. The rules are applied in order of their hierarchical dependence up to the top level of the hierarchy. For each recognized instance, GNS executes an action if one has been provided by the user. The action is written in interpreted C code to which the identified values of the rule generics are passed as parameters. The nature of the action is evidently highly customizable, but an obvious application is the generation of a behavioral description customized to the identified structure.

3.2. Execution Modes

In general, the GNS command is used as follows :

```
yagle -gns [options] input_name
```

GNS reads the transistor net-list given by input_name. If input_name corresponds to the top level of a hierarchical net-list, then this net-list is first flattened. The hierarchical recognition is then performed until all the rules have been exhausted. The netlist is then partitioned into recognized and non-recognized parts. The standard Yagle automatic functional abstraction is performed on the non-recognized part, with all options of Yagle being taken into account. A VHDL or Verilog description for this is generated with the name <input_name>_yagcore, in a addition a structural Netlist is generated (in the format specified by avtOutputNetlistFormat) regrouping the recognized and non-recognized parts with the name <input_name>_yagroot. However, using the behavioral API (beg_API), it is possible to obtain a single file <input_name>.vhd.

GNS can also be executed as :

```
yagle -xg [options] input_name
```

In this mode execution terminates after the recognition phase and the transistor netlist of the non-recognized parts of the circuit is generated (in the format specified by avtOutputNetlistFormat) instead of the behavioral description.

3.3. Options Available

All the options available to Yagle are available to GNS to control the functional abstraction of the unrecognized parts.

3.4. Output Files

<input netlist>_yagroot.<vhd|v|spi>
: This file contains a structural description regrouping the blocks recognized by the GNS module and the remainder of the transistor netlist.

<input netlist>_yagcore.<vhd|v|spi>
: Depending on the execution mode, this file contains either an automatically abstracted behavioral model of the unrecognized remainder of the transistor netlist, or the transistor netlist itself.

<input netlist>.<vhd|v>
: If all the recognized blocks are given a behavior using the behavioral API (beg_API), this file will replace the two previous and will contain the merge of all the behaviors including the behavior automatically generated by YAGLE.

Chapter 4. Defining Recognition Rules and Actions

4.1. Description

GNS provides a highly efficient way to identify a hierarchical structure within a flat transistor netlist. This process is done by first identifying basic blocks made up exclusively of transistors. Those blocks are then merged by hierarchical rules to a higher level of abstraction. The process is repeated until all the user specified hierarchical rules have been fully exhausted. Each user-specified rule is a model corresponding to a single hierarchical level. Genius will try to find all the instances of each level starting from the models describing the transistor level blocks right up to the top-level hierarchical model. In addition, the user is given the possibility of specifying actions which are performed whenever instances of a model are identified within the circuit. This action consists of a C or TCL function which can do practically whatever the user wants. The operation of the GNS hierarchical recognition therefore requires a certain amount of user-supplied information and files. To provide this information the user must create a GNS library file which basically contains the file names of the models and actions.

4.2. The Recognition Models

4.2.1. The Transistor Level Models

The first step in the hierarchical recognition is to identify the blocks at the lowest level. Those blocks are composed exclusively of transistors. The user must therefore provide models for each of the basic blocks to recognize. The actual recognition of these transistor-level blocks is performed by FCL, however this is transparent to the user since the GNS module handles the communication with FCL. The description of the transistor level blocks can be given either as a SPICE netlist or in a structural VHDL file.

The spice netlist is a classical flat transistor netlist but the VHDL description is done by instantiating transistors N or P. For this, there are two special components to describe the transistors:

```
COMPONENT tn
   PORT (
   grid: IN BIT;
   source, drain: INOUT BIT);
END COMPONENT;

COMPONENT tp
   PORT (
      grid: in bit;
      source, drain: inout bit);
END COMPONENT;
```

From the GNS point of view, there is no difference between these representations. For instance, an inverter could be described as:

In VHDL:

```
ENTITY inverter IS
   PORT (
   dout: out bit;
   din: in bit;
   vdd, vss: in bit);
END inverter;

ARCHITECTURE structural OF inverter IS

   COMPONENT tn
      PORT (grid: IN bit; source, drain: INOUT bit);
   END COMPONENT;

   COMPONENT tp
      PORT (grid: IN bit; source, drain: INOUT bit);
   END COMPONENT;

BEGIN
   tn_1: tn
      PORT MAP (grid=>din, source=>vss, drain=>dout);
   tp_1: tp
      PORT MAP (grid=>din, source=>vdd, drain=>dout);
END structural;
```

Or in SPICE:

```
.SUBCKT inverter dout din vdd vss
M1 vdd din dout vdd tp
M2 vss din dout vss tn
.ENDS
```

The user has the freedom to use whichever format he/she prefers. The library file is used to specify the format to be used for a particular model.

4.2.2. The Hierarchical Models

Once the basic transistor-level blocks have been identified within the circuit, the higher-level hierarchical models can be identified. In order to perform this hierarchical recognition, the user must specify, for each of the hierarchical levels, how the blocks are interconnected. This is done by providing a standard VHDL structural description of how the lower-level blocks are interconnected.

Structural VHDL provides a natural way of describing a hierarchical structure. The ENTITY construct effectively serves as a declaration for the recognition rule, as well as providing the interface of the component abstracted by application of the rule. The ARCHITECTURE block specifies the interconnection of blocks from the lower levels required by the rule. As in VHDL, the required lower-level blocks must be declared using the COMPONENT construct, this allows both implicit (order based) and explicit (name based) component instantiations. Anyway, using the format spice_hr rather than VHDL permits the description of a simple hierchical model without all the possibilties of the VHDL model (loops, vectors, ...).

For example, the rule required to recognize a bi-stable made up of two NAND gates would be written as follows:

```
ENTITY bistable IS
   PORT (
   r: IN BIT;
   s: IN BIT;
   q: OUT BIT;
   qn: OUT BIT;
   vdd, vss: IN BIT);
END bistable;

ARCHITECTURE nand_based OF bistable IS

   COMPONENT nand
      PORT (y: OUT BIT; in1, in2: IN BIT; vdd, vss: IN BIT);
   END COMPONENT;

BEGIN
   nand_1: nand
      PORT MAP (q, r, qn, vdd, vss);
   nand_2: nand
      PORT MAP (qn, s, q, vdd, vss);
END nand_based;
```

However, it is also possible to generate a bi-stable from two NOR gates. The user may therefore require that a single recognition rule consist of alternative internal structures. This is possible by providing multiple ARCHITECTURE blocks for a single ENTITY.

To recognize a bistable made up of either NAND or NOR gates, the user would add the following architecture construct to the preceding rule:

```
ARCHITECTURE nor_based OF bistable IS

   COMPONENT nor
      PORT (y: OUT BIT; in1, in2: IN BIT; vdd, vss: IN BIT);
   END COMPONENT;

BEGIN
   nor_1: nor
      PORT MAP (q, r, qn, vdd, vss);
   nor_2: nor
      PORT MAP (qn, s, q, vdd, vss);
END nor_based;
```

4.2.3. Generic Hierarchical Models

One of the most powerful features of the GNS recognition language is the ability to represent interconnections of an arbitrary number of identical structures. Both parallel structures and serial structures can be represented.

- Parallel structures are those where an arbitrary number of identical blocks share one or more common signal.
- Serial structures are those where an arbitrary number of identical blocks are connected in cascade.

This important capability is provided for through the use of VHDL Generics coupled with GENERATE constructs. In standard VHDL a structural model can be made generic in exactly this way; it is the responsibility of the instantiation to specify the actual value of the generic variable. In our semantic, it is the role of the recognition module to identify the value.

A typical example of a parallel structure would be a rule to identify a column of an arbitrary number of bit-cells as part of a static RAM:

```
ENTITY column IS
   GENERIC (capacity: INTEGER);
   PORT (
   q, nq: INOUT BIT;
   com: IN BIT_VECTOR (1 TO capacity);
   vdd, vss: IN BIT);
END;

ARCHITECTURE structural OF column IS

   COMPONENT bitcell
      PORT (q, nq: INOUT BIT; com, vdd, vss: IN BIT);
   END COMPONENT;

BEGIN
   loop: FOR i IN 1 TO capacity GENERATE
      bit_i: bitcell
         PORT MAP (q, nq, com (i), vdd, vss);
   END GENERATE;
END structural;
```

- The generic variable must be declared in the ENTITY block.
- A GENERATE loop must be defined using the generic variable to define the upper limit.
- One or more signals or external connectors must be vectorized using the value of the generic variable to define the range.

Both the loop limits and the vector bounds can be specified as expressions. In this case, the expression for the lower limit must necessarily resolve to a constant and for the upper limit, the generic variable must be the only unknown.

In any case, in any one rule, it is not possible to have more than one GENERATE loop for which the limits are specified by a GENERIC whose value is to be identified. However, this is not a real limitation thanks to the unlimited hierarchy of the rules.

It is also possible to describe arbitrary numbers of components connected in cascade, for example, a chain of inverters can be represented by the following rule:

```
ENTITY invchain IS
   GENERIC (length: INTEGER);
   PORT (
   q, nq: INOUT BIT;
   con: IN BIT_VECTOR (0 TO length);
   vdd, vss: IN BIT);
END;

ARCHITECTURE structural OF invchain IS

   COMPONENT inverter
```

```
      PORT (y: OUT BIT; a, vdd, vss: IN BIT);
   END COMPONENT;

BEGIN
   loop: FOR i IN 1 TO length GENERATE
      inv_i: inverter
         PORT MAP (con (i), con (i-1), vdd, vss);
   END GENERATE;
END structural;
```

4.2.4. Exploiting Generic Variables

At each hierarchical rule level it is possible to exploit the value of generic variables obtained from lower levels of the hierarchy. The mechanism for this is identical to standard VHDL apart from the fact that the direction of transmission is from the lower levels upwards rather than from the upper levels downwards.

The transmission of generic values is specified using the GENERIC MAP construct. This construct specifies which generic variable in a rule takes the value of a generic variable defined in rule corresponding to an instantiated model.

For example, imagine we need the height of a column of bit cells in order to add an inverter in front each of the command inputs. We could use the following rule:

```
ENTITY buffered_col IS
   GENERIC (height: INTEGER);
   PORT (
      q, nq: INOUT BIT;
      bufcomb: IN BIT;
      vdd, vss: IN BIT);
END bistable;

ARCHITECTURE structural OF buffered_col IS

   COMPONENT column
      GENERIC (capacity: INTEGER);
      PORT (
         q, nq: INOUT BIT;
         com: IN BIT_VECTOR (1 TO capacity);
         vdd, vss: IN BIT);
   END COMPONENT;

   COMPONENT inverter
      PORT (out: OUT BIT; a: IN BIT; vdd, vss: IN BIT);
   END COMPONENT;

   SIGNAL bufcom: BIT_VECTOR (1 TO height);

BEGIN
   column_ins: column
      GENERIC MAP (height=>capacity);
      PORT MAP (q, nq, bufcom, vdd, vss);

   loop: FOR i IN 1 TO height GENERATE
      buf_i: inverter
         PORT MAP (bufcom (i), bufcomb (i), vdd, vss);
   END GENERATE;
```

```
END structural;
```

As we can see from the above example, in order to exploit a generic variable whose value obtained by a rule at a lower level, is is necessary to:

- Declare the GENERIC in the COMPONENT corresponding to the lower-level rule.
- Supply the appropriate GENERIC MAP with the instantiation of the model corresponding to the lower-level rule.

The GENERIC MAP can be either explicit (as in the above example) or implicit, in which case the association is made based upon the order of the GENERIC declarations in the COMPONENT.

Generic variables whose values are obtained through GENERIC MAPs can be used in the same way as generic variables identified in the rule. That is for:

- Range declarations in signals and/or connectors.
- Loop limits in GENERATE statements.

A rule can contain any number of GENERATE statements defined by GENERIC variables obtained in this way, since their limits are precisely defined at the beginning of the validation of the rule.

4.3. VHDL Recognition Rules Reference

This section contains the complete grammar of the structural VHDL subset used to represent the GNS recognition rules. Only entity and structural architecture declarations are required. Before giving the BNF for each of these two parts, we illustrate them with a general example.

The legend for the following BNF grammar definitions are:

<name>	is a syntax construct item
name	is a lexeme
[<name>] or [name]	is an optional item
<name>*	is zero or more items
<name>+	is one or more items
<name> ->	indicate a syntax definition to an item
||	introduces an other syntax definition for an item

First of all, each rule requires an entity declaration such as:

```
ENTITY levelA IS
   GENERIC ( VAR, VAR1, ...: integer );
   PORT (
      a INOUT BIT;
      b: IN BIT_VECTOR(1 TO var);
      ...
      vdd, vss: IN BIT);
END;
```

The entity describes the model "levelA". A model can be instantiated with parameters. All the parameters used later in the architecture part of the model must be present in the model entity. One, and only one, of the generic variables can be the special variable whose value must be determined during execution of the rule.

The complete entity grammar allowed is as follows:

```
<entity> -> ENTITY <name_of_model> IS
            [GENERIC ( <list_of_variables>+ );]
            PORT ( <list_of_ports>+ );
            END [<name_of_model>];

<list_of_variables> -> <<variable_decl>;>* <variable_decl>

<variable_decl> -> [VARIABLE] <<identifier_list>: INTEGER

<identifier_list> -> <<identifier>,>* <identifier>

<list_of_ports> -> <<port_decl>;>* <port_decl>

<port_decl> -> <identifier_list>: <port_type>
```

```
<port_type> -> IN <mode>
            || OUT <mode>
            || INOUT <mode>

<mode> -> BIT
       || BIT_VECTOR <array>

<array> -> ( <expression> TO <expression> )
        || ( <expression> DOWNTO <expression> )
```

The expression in the array statement is a standard arithmetic expression which can depend on all the generic variables defined in the entity, even the special generic variable whose value must be identified during execution of the rule.

The architecture section of a rule is composed of 2 parts: the component declarations and the model instantiations in the architecture body. The component of a model should have the same interface and the same generic variables found in the entity statement for the same model. The grammar of the architecture body is elementary.

A typical example shows loops instantiating multiple instances and single instantiations:

```
architecture ArchiLevelA of levelA is

COMPONENT levelB1
   GENERIC (varB2, ...: integer)
   PORT ( ...
      q: IN BIT_VECTOR(1 TO varB2);
      ...
      vdd,vss: IN BIT);
END COMPONENT;

BEGIN
   loop0: FOR i IN 1 TO var GENERATE
      instB1_i: levelB1
         GENERIC MAP (varB2=>var1, ...)
         PORT MAP ( ... );
   END GENERATE;

   loop1: FOR j IN 1 TO var1*2 GENERATE
      instD1_j: levelD1
         GENERIC MAP ( ... )
         PORT MAP ( ... );
   END GENERATE;

   instance_fcl: levelFCL1 GENERIC MAP (...) PORT MAP ( ... );
   instance_l2: levelC PORT MAP ( ... );
END;
```

The complete architecture grammar allowed is as follows:

```
<architecture> ->
        ARCHITECTURE <name_of_archi> OF <name_of_model> IS
        <component>*
        <signal_decl>*
        <architecture_body>*

<signal_decl> -> SIGNAL <identifier_list>: <mode>;

<component> -> COMPONENT <name_of_model>
```

```
                [GENERIC ( <list_of_variables>+ );]
                PORT ( <list_of_ports>+ );
                END COMPONENT;

<architecture_body> -> BEGIN
                       <architecture_element>*
                       END [<name_of_archi>];

<architecture_element> -> <simple_instantiation>
                       || <loop_instantiation>

<loop_instantiation> ->
        <blockname>:
        FOR <identifier> IN <expr> <direction> <expr>
        GENERATE
        <instantiation>+
        END GENERATE;

<simple_instantiation> ->
        <blockname>: <identifier>
        [GENERIC MAP ( <generic_assignment> )]
        PORT MAP ( <port_assignment> );

<direction> -> TO
            || DOWNTO

<instantiation> -> <loop_instantiation>
                || <simple_instantiation>

<generic_assignment> -> <identifier_list>
                     || <explicit_assignment_list>

<explicit_assignment_list> ->
        <<explicit_assignment>,>* <explicit_assignment>

<explicit_assignment> -> <identifier> => <identifier>

<port_assignment> -> <signal_list>
                  || <explicit_sig_assign_list>

<signal_list> -> <<signal>,>* <signal>

<signal> -> <identifier> [<array>]

<explicit_sig_assign_list> ->
        <<explicit_sig_assign>,>* <explicit_sig_assign>

<explicit_sig_assign> -> <identifier> => <signal>
```

The instantiation of a model is done by assigning the generic variable of the instance to the local generic variables of the current model and by linking the instance port to the model signals. The <identifier> is the model name and <blockname> the instance name.

The user can also use a "for" statement to try to instantiate a set of instances whose number is known. In this case, the expressions defining the bound of the "for" statement have no restriction and can depend on any of the known generic variables.

The user may also wish to recognize an unknown number of instances such as cells in parallel or in series. In this case, the left or right expression in the "for" statement contains an unknown generic variable and at least one of the port vector will depend on this variable. The expression with the generic variable has certain restrictions. The only legal operators are '+', '-', '/' and '*'.

When trying to find a number of cells in a "for" statement, Genius will always try to match as many instances as possible.

4.4. The Actions

The user can associate an action to a model. This action can be put directly at the end of the VHDL file if it's a C action or in a separate file. If the action is written in TCL it can be defined in the main TCL script. The action is a C or TCL function which has the same name as the model. This function is given the values computed for all the generic variables of the model. Those values are assigned to variables with the same name as the generic variables of the model.

For any given model, there can be several versions due to the fact that each recognized instance of a model can differ from another depending on the values attributed to the generic variables used to match the model. A unique name for each version is passed as an argument to the function as a string "char *model".

The action is called for each recognized instances remaining after all the recognition rules have been applied. A unique name for each instance is passed as an argument to the function "char *instance".

```
void levelA (char *model, char *instance, int VAR, int VAR1, ...)
{
 ...
}
```

In TCL, generic variables are made available as global TCL variables. Be aware that calling TCL script within GNS, affects main TCL script.

```
proc levelA {} {
  global model instance VAR VAR1 ...
  ...
}
```

The actions are called at the end of the complete recognition phase for all the instances to keep (see Library File) in the circuit. The instances to keep but only used by an higher hierarchical level are destroyed. Their actions will therefore not be executed.

The C action function is interpreted by GNS, therefore there are certain limitations in what the user can do. However, the action functionality can be expanded by the use of dynamic libraries. Dynamic libraries, containing user-defined functions which can be called from within actions, are easily created with the help of the genapi tool.

4.4.1. Types

The action interpreter authorizes use of the following base types:

- char

- int
- double
- FILE *

Pointers types or one dimensional arrays types based on the above type ar also recognized.

Additionally, any other pointer types can be specified so long as they are only used in user-defined function calls (see dynamic library).

Static variables are authorized. They are global to all the actions.

4.4.2. Supported Operators

All the basic C arithmetic and logic operations are possible on the authorized interpreter types.

4.4.3. Functions

The interpreter can manage the following standard C functions:

- printf
- fprintf
- sprintf
- malloc
- free
- strcpy
- strcat
- fopen
- fclose

The following built-in non standard functions are also implemented:

FILE *avtfopen(char *name, char *extension, char mode) ....
: opens a file taking into account the global configuration file for the seek directories and compression filter. Legal values for the mode are: READ_TEXT and WRITE_TEXT

int gnsModelVisited(char *model);
: Returns 1 if the given model has already been marked as visited, 0 otherwise. Used to avoid duplicating actions.

void gnsMarkModelVisited(char *model);
: Marks the given model as having been visited, to avoid duplicating the action.

char *char_to_string(int size, char caract);
: returns a string which length is 'size'. The string is filled with the character 'caract'.

char *onehot_to_bit(int size, int bitnum);
: returns a string which length is 'size'. The string is filled with the value (1<<bitnum) in binary format.

char *onehot_to_hexa(int size, int bitnum);
returns a string which length is 'size'. The string is filled with the value (1<<bitnum) in hexadecimal format.

char *onehot_to_octa(int size, int bitnum);
returns a string which length is 'size'. The string is filled with the value (1<<bitnum) in octal format.

char *onecold_to_bit(int size, int bitnum);
returns a string which length is 'size'. The string is filled with the value not(1<<bitnum) in binary format.

char *onecold_to_hexa(int size, int bitnum);
returns a string which length is 'size'. The string is filled with the value not(1<<bitnum) in binary hexadecimal format.

char *onecold_to_octa(int size, int bitnum);
returns a string which length is 'size'. The string is filled with the value not(1<<bitnum) in octal format.

char *genius_date();
returns a string containing the current date and time.

Note that each of the string generation functions always returns a pointer to the same address. If you wish to keep the string over multiple calls of the same function, you will have to copy the result into another string variable.

4.4.4. Loops and conditional statements

Genius interpreter handle the following C loop and conditional constructs:

- for (...; ...; ...) {...}
- do {...} while (...)
- while (...) {...}
- if (...) {...} else {...}

The reserved words "return" and "break" are also supported as well as the standard function call "exit(<errcommand>)".

4.4.5. Dynamic Libraries

The user can create his own dynamic library whose functions can be called from within GNS actions. To do so, he has to provide a simplified header file containing the function prototypes. Global variables are not allowed. The use of the tool genapi will automatically generate the dynamic library from the header file and the .c files given by the user.

4.5. The Library File

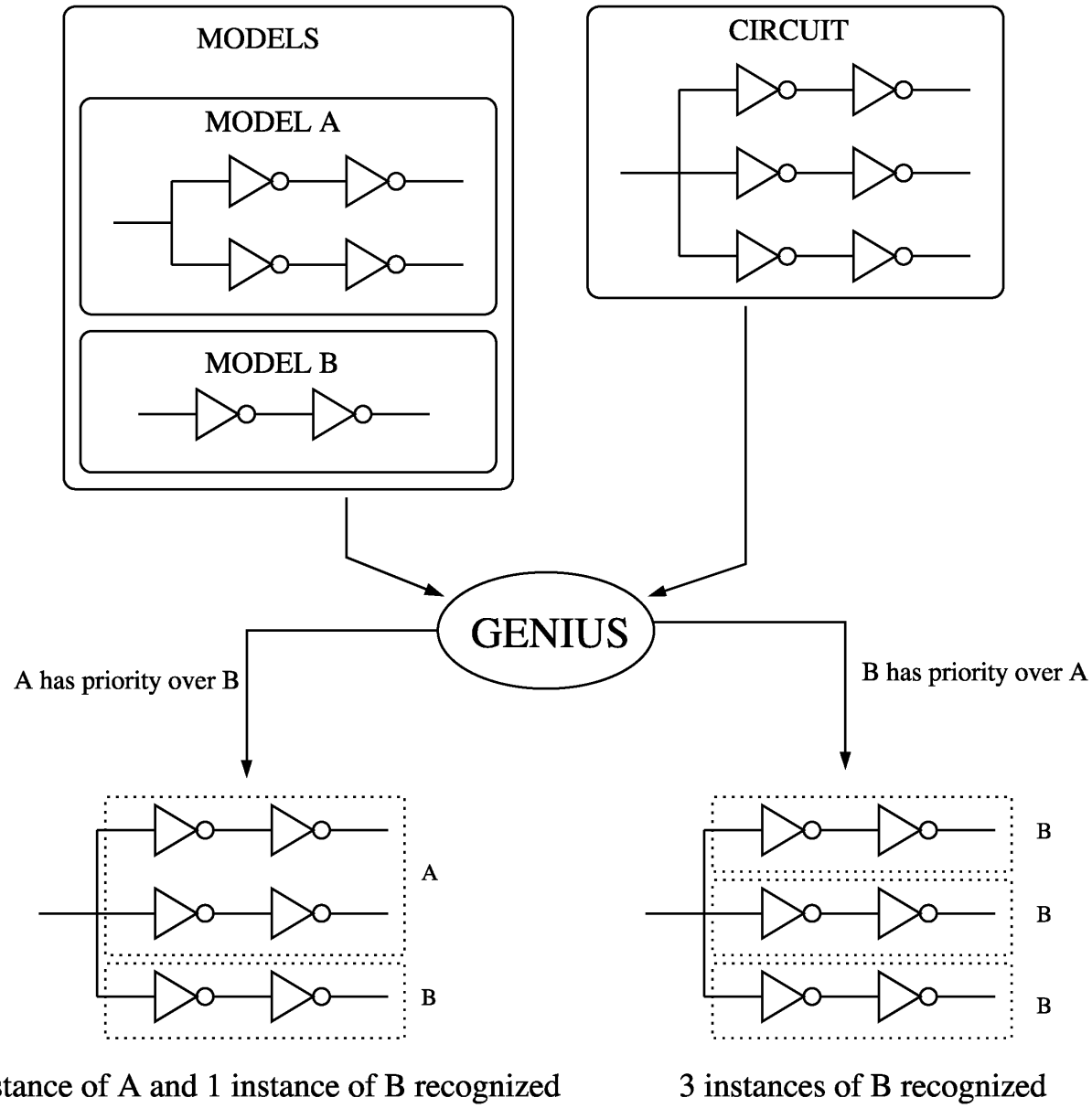
The library file contains a list of all the rule files and supplementary information required by the recognition module. The transistor level models, the hierarchical models and the action files are all referenced here. The library file can also indicate built-in template recognition rules and actions to use in the recognition process.

Each line of the library file can have two format. The first one is for the user defined rules and actions and the second one for the template instantiations:

```
<model_file> [: [priority=<num>]
                [, keep=<yes|no>]
                [,format=<spice|vhdl|spice_hr>] ];
```

The names of a model_file does not have to respect a particular format, except if the file is supposed to be a TCL script. In this case the file must be interpreted by TCL the suffix must be ".tcl" else the file is considered to be VHDL or C. The format directive specifies whether a transistor-level recognition rule is in SPICE or VHDL format. In the case of files containing actions, the format must not be specified.

The priority defines the order of recognition for models. Lower value means higher priority. Independent recognition rules are applied in order of specified priority. This option can be critical if one model is a subset of another model.



1 instance of A and 1 instance of B recognized

3 instances of B recognized

Finally, the "keep" flag tells genius whether the user is interested in keeping the recognized instances of a model. The user is generally interested by the top level of his hierarchy but he can keep other recognized instances (see also gnsKeepAllCells configuration). The <modelname>_yagroot.<vhd|spi> file will contain only the instances whose model keep flag have been set.

In the case of template instantiations, the syntax follows:

```
<model_name> <model_instance_name> {
  [ rule=<file_spec> ]
  [ rules={ <file_spec> [, ...] } ]
  [ action=<file_spec> ]
  [ actions={ <file_spec> [, ...] } ]
  [ <model_identifier>=<new_identifier> [, ...] ]
} [: [priority=<num>]
      [, keep=<yes|no>]  ];
```

The model_name is the name of one of the built-in templates available in the distribution or in the paths given by 'gnsTemplateDir' and 'gnsLibraryDir' variables. The model name can be overridden by model_instance_name but the model_name can be used if the template is instantiated once.

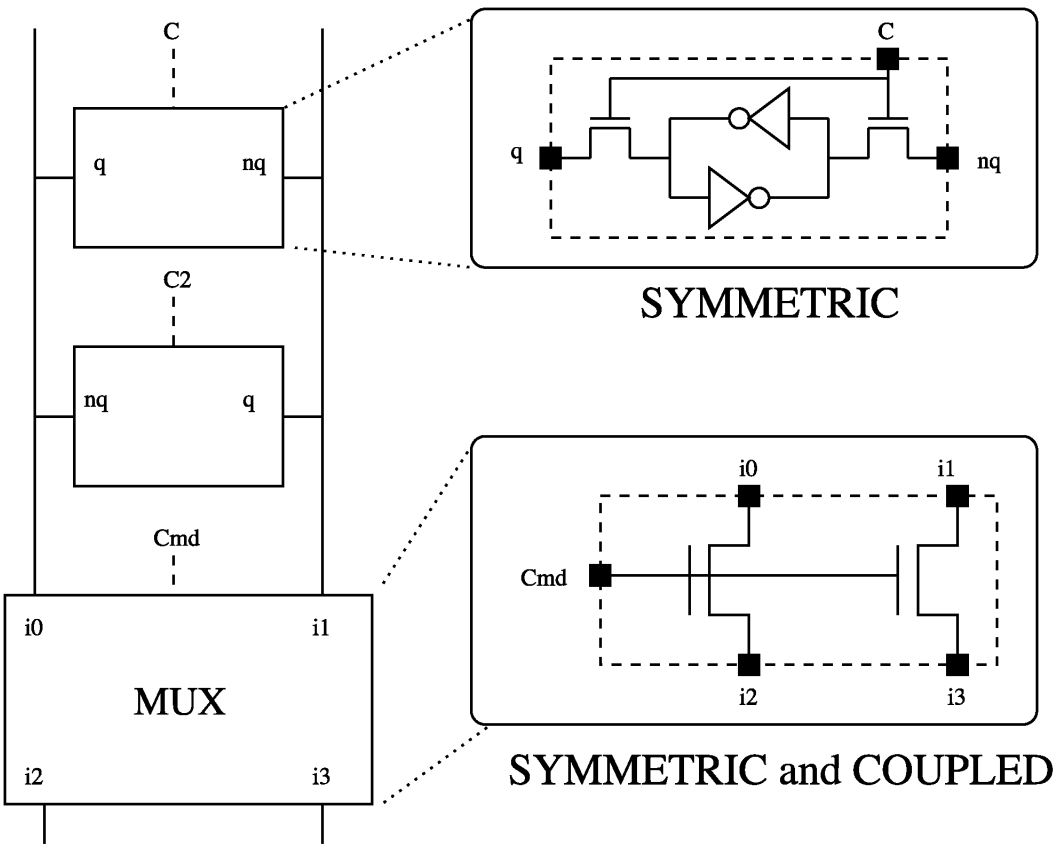
By default, the template rules and actions will be used for the recognition but they can also be overridden by specifying a <file_spec> which has exactly the same syntax as the user defined rule except the trailing semicolon. Several files can be specified for the rules and the actions, for instance, if different architectures are spread over multiple files.

All the identifier <model_identifier> of the template rules can be renamed by <new_identifier> depending on the user wish. This is handy to change the instance models, connector names or generic variable names in the rules.

Finally, an overall priority and keep flag can be specified for the instantiated model. Their meaning is the same as for the user defined rules and actions.

4.6. Symmetry and Coupling

Due to the nature of bottom up recognition, any symmetry in the identified components can cause recognition difficulties which are not immediately obvious. Cells are identified at a particular hierarchical level without knowing how they will be connected in the higher level. Because of this, GNS cannot guarantee the precise ordering of any symmetric cell connectors. In the example below representing the memory cell, the connectors q and nq are symmetric. This means that from the perspective of the individual cell there is nothing to distinguish between these two connectors. When performing the recognition of the memory cell, There is nothing to prevent the memory cell being recognized the other way round to that which the user would have expected.



The user must pay careful attention to any possible symmetry in the models to recognize. For example, if the user wants to recognize a column of memory cells connected in parallel, he would provide a rule to identify an arbitrary number of memory cells with the q ports connected together and the qn ports connected together. However, if q and qn ports are symmetric then q ports may be connected to qn ports and vice versa but nonetheless corresponding to a column of memory cells.

The case of symmetry also occurs often when dealing with vectors. A user may require the identification of two instances connected together by vectored ports. However, the signal connected to bit(0) of one of the ports is not necessarily the signal connected to bit(0) of the other port. The order of the indexing of the vectored ports depend on the order of the recognition algorithm.

To cope with this problem of symmetry, GNS provides for a mechanism of pragma directives included as comments in the model files. Each pragma directive consists of the list of symmetric connectors. The connectors can be single connectors or the radical of a vectored connector. Use of the radical of a vector implies that all the connectors of the vector are symmetric to each other. Each group of symmetric connectors must be declared in a unique pragma.

In VHDL rules, the pragma is declared in the model entity after the port declarations as follows:

```
-- pragma symmetric [<single_connector>|<connector_radical]+
```

In SPICE, for symmetric connector in transistor-level cells, the pragma can be placed anywhere as follows:

```
* pragma symmetric [<single_connector>|<connector_radical]+
```

With symmetry comes the additional difficulty of coupled symmetry. When GNS tries to match together recognized instances with symmetric connectors and the initial match fails, it only knows that there are connectors which can be swapped. It will then try to swap around the symmetric connectors until a match is found. In reality, the symmetry handling is more efficient than this, but that's the basic idea.

In the above example of a column multiplexer, there are 2 sets of symmetric connectors:

- i0 is symmetric with i1
- a0 is symmetric with a1

In the example i0 depends on a0 and i1 depends on a1. It's obvious that if i0 and i1 are swapped, a0 and a1 must also be swapped. a0 is effectively coupled with i0 and a1 is coupled with i1. If the coupling dependencies are not explicitly stated, GNS may swap the connectors in one of the symmetric sets but not the other, thus corrupting the connectivity of the recognized instance.

The user must therefore also be careful about the coupled symmetric connectors in a model else GNS will be able be match instances for which the connectivity described by the rule is not respected.

In the same way as for standard symmetric connectors, pragma directives must be used to define the couplings:

In VHDL rules, the pragma is declared in the model entity after the port declarations as follows:

```
-- pragma coupled [<single_connector>|<connector_radical]+
```

In SPICE, for symmetric connector in transistor-level cells, the pragma can be placed anywhere as follows:

```
* pragma coupled [<single_connector>|<connector_radical]+
```

The entity declaration for the multiplexer would therefore be as follows:

```
ENTITY mux IS
   PORT (
      i0, i1: INOUT BIT;
      a0, a1: INOUT BIT
   );
-- pragma symmetric i0 i1
-- pragma symmetric a0 a1
-- pragma coupled i0 a0
-- pragma coupled i1 a1
END;
```

The user should be aware that symmetric connector imposes a restriction on how the rules can be written. If a particular rule contains a loop instantiation (of the special kind which determines the value of a generic) and the instances in the loop are connected together by symmetric connectors, then is not possible, in the same rule, to connect these connectors to anything outside the loop apart from a set of symmetric connectors.

This is not actually a restriction on what can be recognized since structures which require this kind of description can always be handled by introducing an additional rule hierarchy.

4.7. Other PRAGMAs

```
-- pragma without <name0> [<name1]*
```

Matches the hierarchical recognition rule only in no correspondance is found for the given instances or transistors.

```
-- pragma exclude <name0> [<name1]*
```

After the each recognition of the rule, the given instances are put back in the available instance list so they can be used again in the recognition rules (even the current one). This permits sharing of instances/transistors between rules.

```
-- pragma exclude_at_end <name0> [<name1]*
```

The same as "exclude" but the given instances can not be reused by the current rule. They will be available for the next ones.

```
-- pragma forcematch <name0> [<name1]*
```

The given names apply to transistor, instances and net names. This option, will match the names in the rule if they are in the given list. Name match for instances won't work for GNS recognized instances as their name is automatically generated. Instead blackboxed instances can use this option. If '_' is used in a given name, it can be matched with a hierarchy separator. eg. circuit blackbox instance xlevel.xcells will be matched by xlevel_cells.

```
-- pragma unused <name0> [<name1]*
```

Each connector in the given list is unused meaning there is no real correspondance to them in the netlist.

Chapter 5. Extending GNS with Dynamic Libraries

5.1. Overview

The Avertec set of verification tools provide a complete platform for the verification of electrical and functional aspects at the back-end of a design flow. The standard tools in themselves provide a means of performing most of the standard verification requirements.

However, users with particular verification flow requirements may require a higher degree of customizability of the various technology modules. For example, the GNS hierarchical recognition module allows for the execution of operations upon recognized objects, the user way require that certain complex verification operations be performed on these objects involving other Avertec modules, or even external tool.

The GNS provides a high-level programmatical interface to a growing number of the Avertec technology modules providing, in combination with GNS, a powerful mechanism for generating turnkey verification flows for custom components. In addition, a mechanism is provided for the user to dynamically link his own verification code directly into the Avertec platform.

5.2. Description

The GNS is a set of dynamic libraries which act as high-level APIs to the complete Avertec verification platform. A growing number of the components which make up the Avertec tools have a corresponding dynamic library API. These dynamic libraries contain functions designed to be called from within GNS recognition actions (see GNS User Guide) or for the `avt_shell` Tcl script interface to the technology modules.

In addition, it is also possible for the user to create his own APIs. The GNS provides a simple method for user code to be transformed into a shared library for direct use from within GNS.

5.3. Integrating the APIs in an Avertec Tool Flow

The APIs are available whenever the GNS recognition module is activated. Currently, this is possible in the Yagle, GNS and TMA tools. The activation of the GNS module by the appropriate options results in the loading of all the dynamic libraries specified in the configuration file. The functions defined in these libraries are then available for use in user-defined GNS actions.

The set of APIs provided as standard with the Avertec tools are documented in subsequent chapters of this guide. The include, amongst others, functions to interrogate the recognition database, perform 'in situ' SPICE simulations to calculate delays and timing constraints, build behavioral models and build timing models.

Chapter 6. Creating a User-Defined Dynamic Library API

6.1. Description

It is an extremely simple task for the user to generate his own APIs which can be dynamically linked into an Avertec verification flow. In order to do this, we provide the genapi tool which, from a set of C source files, together with header files declaring the visible functions, creates a shared library which can be used in exactly the same way as the supplied APIs.

6.2. Executing the Genapi Tool

To generate your own dynamic library for linking with the Avertec tools, you should use the Genapi tool as follows:

```
genapi <f.c g.c ...> -i <interface.h> [-o <lib.so>] [--keep_files] [-kf]
```

This takes the specified set of C source code files, together with a header file declaring the visible functions and generates directly a shared library compatible with GNS actions. The header file must contain full ANSI prototype declarations, since this is used to create wrapper functions for the functions to be made visible. By default the shared library generated (file with ".so" suffix) has the same base name as the header file but this can be modified using the "-o" option. The "-kf" (or "--keep_files") option prevents the removal of the intermediate files generated, such as the source code for the wrapper functions and the Makefile.

Chapter 7. API Functions Available

7.1. GNS Built-in

The gen_builtin_funtions API provides a set of basic functions useful in GNS actions which need to generate output files.

7.1.1. char_to_string

```
char *char_to_string(int size, char caract)
```

char_to_string create a string which length is 'size'. The string is filled with the caracter 'caract'.

7.1.2. onehot_to_bit

```
char *onehot_to_bit(int size, int bitnum)
```

onehot_to_bit(size, bitnum) returns a binary string whose length is 'size'. The string represents a binary value where bit number 'bitnum' is set to 1 and the others to 0. The 'bitnum' is little endian oriented.

7.1.3. onehot_to_hexa

```
char *onehot_to_hexa(int size, int bitnum)
```

onehot_to_hexa(size, bitnum) returns a hexadecimal string whose length is 'size'. The string represents a hexadecimal value where bit number 'bitnum' is set to 1 and the others to 0. The 'bitnum' is little endian oriented.

7.1.4. onehot_to_octa

```
char *onehot_to_octa(int size, int bitnum)
```

onehot_to_octa(size, bitnum) returns an octal string whose length is 'size'. The string represents an octal value where bit number 'bitnum' is set to 1 and the others to 0. The 'bitnum' is little endian oriented.

7.1.5. onecold_to_bit

```
char *onecold_to_bit(int size, int bitnum)
```

onecold_to_bit(size, bitnum) returns a binary string whose length is 'size'. The string represents a binary value where bit number 'bitnum' is set to 0 and the others to 1. The 'bitnum' is little endian oriented.

7.1.6. onecold_to_hexa

```
char *onecold_to_hexa(int size, int bitnum)
```

onecold_to_hexa(size, bitnum) returns a hexadecimal string whose length is 'size'. The string represents a hexadecimal value where bit number 'bitnum' is set to 0 and the others to 1. The 'bitnum' is little endian oriented.

7.1.7. onecold_to_octa

```
char *onecold_to_octa(int size, int bitnum)
```

onecold_to_octa(size, bitnum) returns an octal string whose length is 'size'. The string represents an octal value where bit number 'bitnum' is set to 0 and the others to 1. The 'bitnum' is little endian oriented.

7.1.8. genius_date

```
char *genius_date()
```

genius_date() return a string containing the current date and time.

7.1.9. gns_ModelVisited

```
int gns_ModelVisited(char *name)
```

gns_ModelVisited(<name>) returns 0 if the model <name> has been set as visited thru the function gns_MarkModelVisited.

7.1.10. gns_MarkModelVisited

```
void gns_MarkModelVisited(char *name)
```

gns_MarkModelVisited(<name>) adds <name> in the list of model already visited.

7.2. Transistor Netlist Recognition

The FCL API allows the user to use GNS actions to generate all the transistor netlist markings of the FCL transistor recognition module.

7.3. Available Markings

The markings available for the signals are the following ones:

NET_LATCH
Signal corresponds to a latch memory-point.

NET_FLIPFLOP
Signal corresponds to a flip-flop memory-point.

NET_MASTER
Signal corresponds to the master memory-point of a flip-flop.

NET_SLAVE
Signal corresponds to the slave memory-point of a flip-flop.

NET_MEMSYM
Signal corresponds to one side of a symmetric memory.

NET_RS
Signal corresponds to one side of an RS bistable.

NET_VDD
Signal corresponds to an alimentation.

NET_VSS
Signal corresponds to ground.

NET_BLOCKER
No branch of a cone (see man gns) can go through the signal.

NET_STOP
Cannot exploit logic beyond this point for functional analysis in the disassembler.

NET_BYPASS
Signal cannot appear in a timing path.

NET_MATCHNAME
The signal is only matched if the name in the pattern to recognize and the name in the source netlist are identical.

NET_SENSITIVE
Marks the signal as a particularly sensitive signal. If a timed behavioral model of this signal is produced then the most precise (but cumbersome) model will be generated.

The markings available for the transistors are the following ones:

TRANS_BLEEDER
Transistor corresponds to a bleeder.

TRANS_FEEDBACK
Transistor corresponds to a feedback transistor of a memory-point.

TRANS_COMMAND
Transistor corresponds to a command transistor of a memory-point, i.e driven by command signal.

TRANS_NOT_FUNCTIONAL
Transistor should be ignored when calculating gate functionality.

TRANS_BLOCKER
No branch of a cone can contain this transistor unless it is the first transistor of the branch.

TRANS_UNUSED
No branch of a cone can contain this transistor.

TRANS_SHORT
The transistor is considered short-circuited, the gate signal no longer contributes to the list of inputs.

TRANS_MATCHSIZE
The transistor is only matched if the dimensions correspond exactly or to within a given tolerance (see FCL configuration).

TRANS_SHARE
The transistor can be matched by several patterns.

7.3.1. fclMarkCorrespondingSignal

```
int fclMarkCorrespondingSignal(char *signame, char *marks)
```

fclMarkSignal marks a signal in the circuit according to the given marking string.

7.3.2. fclMarkCorrespondingTransistor

```
int fclMarkCorrespondingTransistor(char *transname, char *marks)
```

fclMarkTransistor marks a transistor in the circuit according to the given marking string.

7.3.3. fclOrientCorrespondingSignal

```
void fclOrientCorrespondingSignal(char *signame, int level)
```

fclOrientSignal orients a signal according to the given output level.

7.3.4. fclCmpUpConstraint

```
void fclCmpUpConstraint(StringList *siglist)
```

Specifies that one and only one of the signals in the list can be high.

7.3.5. fclCmpDnConstraint

```
void fclCmpDnConstraint(StringList *siglist)
```

Specifies that one and only one of the signals in the list can be low.

7.3.6. fclMuxUpConstraint

```
void fclMuxUpConstraint(StringList *siglist)
```

Specifies that at most one of the signals in the list can be high.

7.3.7. fclMuxDnConstraint

```
void fclMuxDnConstraint(StringList *siglist)
```

Specifies that at most one of the signals in the list can be low.

7.3.8. fclAllowShare

```
void fclAllowShare(Transistor *transistor)
```

Specifies that the given transistor can be shared with another transistor level model.

7.4. GNS Recognition

The GEN API provides a set of functions useful in GNS actions which need to obtain correspondence information between the hierarchical netlist generated by the GNS recognition module and the original circuit.

7.4.1. gns_StripNetlist

```
void gns_StripNetlist(Netlist *netlist)
```

Suppresses all unconnected RC networks. This function is very useful after using netlist reduction.

7.4.2. gns_StripNetlistFurther

```
void gns_StripNetlistFurther(Netlist *netlist)
```

*Suppresses all unconnected RC networks *AND* connectors. This function is very useful after using netlist reduction.*

7.4.3. gns_SetLoad

```
void gns_SetLoad(Netlist *netlist, char *connector, double load)
```

Sets an additional capacitance on the connector `connector'. This function should be primarily used for characterization purposes. Further call of gns_SetLoad overrides previous characterization capacitance setting.

7.4.4. gns_FlattenNetlist

```
Netlist *gns_FlattenNetlist(Netlist *netlist, int rc)
```

Flattens the given netlist to the transistor level or to a cell level. The cell levels can be specified by using gns_SetModelAsLeaf(<modelname>) where model name is the cell name. The RC information can be added by specifying the flags whose value can be INTERNAL_RC, IN_RC, OUT_RC, ALL_RC or 0. Those values can be ORed. In most cases, the returned netlist must be freed by the user.

If used in TCL, `flags` *is a list of the different flags.*

7.4.5. gns_FreeNetlist

```
void gns_FreeNetlist(Netlist *netlist)
```

Deletes a Netlist. It should not be called on a netlist obtained with gns_GetNetlist().

7.4.6. gns_AddRC

```
Netlist *gns_AddRC(Netlist *netlist, int rc)
```

Adds the RC information in the current netlist. The flags whose value can be INTERNAL_RC, IN_RC, OUT_RC, ALL_RC or 0 indicates where the RC should be added. Those values can be ORed. In most cases, the returned netlist must be freed by the user.
If used in TCL, `flags` *is a list of the different flags.*

7.4.7. gns_SetModelAsLeaf

```
void gns_SetModelAsLeaf(char *name)
```

Adds the model <name> to the list of models for which the flatten process won't flat the instances to transistor level. In the flattened netlist should appear the instances whose model is <name>. Beware, the leaf model <name> will be taken into account for all the flatten process. To clear the list of leaves, use gns_SetModelAsLeaf(NULL).

7.4.8. gns_ReduceInstance

```
void gns_ReduceInstance(Netlist *netlist, char *ins_name)
```

Reduces an instance to the corresponding capacitances of its interface pins.

7.4.9. gns_KeepInstance

```
void gns_KeepInstance(Netlist *fig, char *ins_name)
```

Keeps an instance from a reducing process.

7.4.10. gns_AddExternalTransistors

```
void gns_AddExternalTransistors(char *str)
```

Allows the user to choice between two possibilities of representation of external transistors connected to an output pin. When the parameter 'str' is set to 'dynamic', capacitances are extracted from the grid, source or drain of the external transistor, and added into the netlist. When the parameter 'str' is set to 'transistor', the external transistors are added into the netlist, and the transistor's connectors that are not connected to the pin, are connected either to 'gnd' or 'vdd', in a way that they always remain non-passant.

7.4.11. gns_ViewLo

```
void gns_ViewLo(Netlist *ptfig)
```

Displays debugging information about the netlist. Information is displayed on stdout.

7.4.12. gns_DriveNetlist

```
void gns_DriveNetlist(Netlist *ptfig, char *format, char *path, char
*name)
```

Drives the netlist in path with the specified format. <format> can be "spice". The filename will be <name>.EXT where EXT depends on the <format> and the figure name in the file will be <name>.

7.4.13. gns_GetNetlist

```
Netlist *gns_GetNetlist()
```

Returns the netlist corresponding with the currently recognized model. PS: this netlist must not be freed.

7.4.14. gns_DuplicateNetlist

```
Netlist *gns_DuplicateNetlist(Netlist *source)
```

Duplicates the netlist <source>. The copied netlist should be freed later.

7.4.15. gns_GetInstanceNetlist

```
Netlist *gns_GetInstanceNetlist(char *name)
```

Returns the netlist corresponding with a recognized instance used in the current hierarchy. <name> can be hierarchical. PS: this netlist should be freed.

7.4.16. gns_GetCorrespondingSignal

```
Signal *gns_GetCorrespondingSignal(char *name)
```

Returns the signal in the circuit corresponding to a signal in the model.

7.4.17. gns_GetSignalName

```
char *gns_GetSignalName(Signal *signal)
```

Returns the name of a signal. When used with a signal in the model, there will not be any information about the signal index range in the returned string.

7.4.18. gns_GetInstanceName

```
char *gns_GetInstanceName(Instance *instance)
```

Returns the name of an instance.

7.4.19. gns_GetInstanceModelName

```
char *gns_GetInstanceModelName(Instance *instance)
```

Returns the name of the model of an instance in the circuit.

7.4.20. gns_GetModelSignalRange

```
void gns_GetModelSignalRange(char *name, int *left, int *right)
```

Returns the index range of a signal in the model. If left = -1, the signal is not a vector. If <left> < <right> then the signal is range is left to right else if <left> > <right> then the signal range is left downto right.

7.4.21. gns_GetModelConnectorList

```
List *gns_GetModelConnectorList()
```

Returns the list of all the interface connectors of the model.

7.4.22. gns_GetInstanceConnector

```
Connector *gns_GetInstanceConnector(Instance *instance, char *name)
```

Returns the connector <name> of <instance> the model.

7.4.23. gns_GetInstance

```
Instance *gns_GetInstance(Netlist *netlist, char *name)
```

Returns the instance <name> present in <netlist>.

7.4.24. gns_GetConnectorCapa

```
double gns_GetConnectorCapa(Connector *lc)
```

Returns the computed capacitance of the connector `lc'. This capacitance is the sum of all capacitances corresponding with the drain, grid or source connectors linked to `lc'. the model.

7.4.25. gns_GetConnectorList

```
List *gns_GetConnectorList(Netlist *netlist)
```

Returns the list of all the interface connectors of the netlist.

7.4.26. gns_GetConnectorDirection

```
char *gns_GetConnectorDirection(Connector *connector)
```

Returns the direction of a connector.

7.4.27. gns_GetConnectorName

```
char *gns_GetConnectorName(Connector *connector)
```

Returns the name of a connector.

7.4.28. gns_GetConnectorSignal

```
Signal *gns_GetConnectorSignal(Connector *connector)
```

Returns the signal linked to the connector.

7.4.29. gns_GetModelSignalList

```
List *gns_GetModelSignalList()
```

Returns the list of all the signal in the model.

7.4.30. gns_IsSignalExternal

```
int gns_IsSignalExternal(Signal *signal)
```

Returns 1 if the signal is linked to a connector in the interface, 0 otherwise.

7.4.31. gns_Vectorize

```
char *gns_Vectorize(char *name, int index)
```

Returns a string containing the name associated with a vector index : name(index).

7.4.32. gns_Vectorize2D

```
char *gns_Vectorize2D(char *name, int index0, int index1)
```

Returns a string containing the name associated with a 2 vector index : name(index1)(index2).

7.4.33. gns_GetInstanceConnectorList

```
List *gns_GetInstanceConnectorList(Instance *ls)
```

Returns the list of all the connectors of an instance.

7.4.34. gns_GetAllCorrespondingInstances

```
List *gns_GetAllCorrespondingInstances()
```

Returns the list of all the instances in the circuit used when matching the model.

7.4.35. gns_GetAllCorrespondingInstanceModels

```
List *gns_GetAllCorrespondingInstanceModels()
```

Returns the list of all the models in the circuit used when matching the model.

7.4.36. gns_GetCorrespondingTransistor

```
Transistor *gns_GetCorrespondingTransistor(char *name)
```

Returns the transistor in the circuit corresponding to a transistor instance in the model.

7.4.37. gns_GetAllCorrespondingTransistors

```
List *gns_GetAllCorrespondingTransistors()
```

Returns the list of all the transistors in the circuit used when matching the model.

7.4.38. gns_GetTransistorGrid

```
Connector *gns_GetTransistorGrid(Transistor *transistor)
```

Returns grid connector of a transistor.

7.4.39. gns_GetTransistorDrain

```
Connector *gns_GetTransistorDrain(Transistor *transistor)
```

Returns grid connector of a transistor.

7.4.40. gns_GetTransistorSource

```
Connector *gns_GetTransistorSource(Transistor *transistor)
```

Returns source connector of a transistor.

7.4.41. gns_GetTransistorType

```
char gns_GetTransistorType(Transistor *transistor)
```

Returns the type of a transistor.

7.4.42. gns_GetTransistorTypeName

```
char *gns_GetTransistorTypeName(Transistor *transistor)
```

Returns the circuit type of a transistor.

7.4.43. gns_GetTransistorParameter

```
double gns_GetTransistorParameter(char *name, Transistor *transistor)
```

Returns the value of a transistor parameter. <name> can be "w", "l", "as", "ad", "ps" or "pd".

7.4.44. gns_GetTransistorName

```
char *gns_GetTransistorName(Transistor *transistor)
```

Returns the name of a transistor.

7.4.45. gns_GetAllTransistorsConnectedtoSignal

```
List *gns_GetAllTransistorsConnectedtoSignal(Signal *signal)
```

Returns the list of all the transistors in the circuit connected to the given signal at the current step of genius recognition.

7.4.46. gns_VectorIndex

```
int gns_VectorIndex(char *name)
```

Returns the index in a signal name. If the signal is not a vector, the value returned is -1.

7.4.47. gns_VectorRadical

```
char *gns_VectorRadical(char *name)
```

Returns the base name of a signal name. The basic action is to remove the index from a vector name.

7.4.48. gns_CreateVhdlName

```
char *gns_CreateVhdlName(char *name)
```

Returns transforms <name> so it is suitable for a VHDL syntaxe.

7.4.49. gns_ChangeInstanceModelName

```
void gns_ChangeInstanceModelName(Instance *instance, char *name)
```

Changes the model name of a recognized instance. If instance is NULL, the new name is applied to the current instance.

7.4.50. gns_GetSignal

```
Signal *gns_GetSignal(Netlist *netlist, char *signame)
```

Retrieves the signal 'signame' in the netlist 'netlist'. The netlist maybe either flat or hierarchical. To retrieve a signal in a hierarchical netlist, one must provide a hierarchical name, i.e containing the successives instances separated with dots. For example, the signal 'ins1.ins2.sig' describes the signal 'sig' in the instance 'ins2', the instance 'ins2' being contained in the instance 'ins1'.

7.4.51. gns_GetConnector

```
Connector *gns_GetConnector(Netlist *netlist, char *con_name)
```

Retrieves the connector 'conname' in the netlist 'netlist'.

7.4.52. gns_GetTransistor

```
Transistor *gns_GetTransistor(Netlist *netlist, char *tr_name)
```

Retrieves the transistor 'con_name' in the netlist 'netlist'.

7.4.53. gns_AWE_GetWorstInstance

```
char *gns_AWE_GetWorstInstance(Netlist *netlist, char *insname, Connector
*lc, double vdd)
```

Retrieves the worst instance (worst AWE delay) which is connected to the connector 'lc'. 'netlist' is the hierarchical netlist. 'insname' is the generic name of the instance to reduce. For example, if the netlist contains "mem_cell.0", "mem_cell.1", "mem_cell.2" ... 'insname' is "mem_cell" 'vdd' is the value of power supply needed to compute AWE delays.

7.4.54. gns_AWE_GetBestInstance

```
char *gns_AWE_GetBestInstance(Netlist *netlist, char *insname, Connector
*lc, double vdd)
```

Retrieves the best instance (worst AWE delay) which is connected to the connector 'lc'. 'netlist' is the hierarchical netlist. 'insname' is the generic name of the instance to reduce.

7.4.55. gns_AWE_KeepBestInstance

```
Netlist *gns_AWE_KeepBestInstance(Netlist *netlist, Netlist *flatnetlist,
char *insname, Connector *lc, double vdd)
```

Keeps the best instance (best AWE delay) which is connected to the connector 'lc', the other instance are reduced. 'netlist' is the hierarchical netlist, 'flatnetlist' is the flattened netlist. 'insname' is the generic name of the instance to reduce. For example, if the netlist contains "mem_cell.0", "mem_cell.1", "mem_cell.2" ... 'insname' is "mem_cell" 'vdd' is the value of power supply needed to compute AWE delays.

7.4.56. gns_AWE_KeepWorstInstance

```
Netlist *gns_AWE_KeepWorstInstance(Netlist *netlist, Netlist
*flatnetlist, char *insname, Connector *lc, double vdd)
```

Keeps the worst instance (worst AWE delay) which is connected to the connector 'lc', the other instance are reduced. 'netlist' is the hierarchical netlist, 'flatnetlist' is the flattened netlist. 'insname' is the generic name of the instance to reduce. For example, if the netlist contains "mem_cell.0", "mem_cell.1", "mem_cell.2" ... 'insname' is "mem_cell" 'vdd' is the value of power supply needed to compute AWE delays.

7.4.57. gns_AWE_GetOrderedInstanceIndex

```
void gns_AWE_GetOrderedInstanceIndex(Netlist *lofig, char *rule,
Connector *connector, int **tab, int *nb)
```

Creates and fills an array with the index of the instances on the signal connected to <connector>. The Instances are ordered with respect to their delay versus connector <connector>. The <rule> defines how to retreive the instance index from the instance name. The '?' in the rule is the number desired. eg. "bitline(?).low(5).latch".

7.4.58. gns_GetInstanceLoopIndex

```
int gns_GetInstanceLoopIndex(Instance *ins, char **ptptname)
```

Returns the index of <instance> and if <radical>!=NULL, fetch <radical> with the name of the instance in the model.

7.4.59. gns_GetInstanceLoopRange

```
void gns_GetInstanceLoopRange(Netlist *lf, Instance *ins, int *left, int
*right)
```

Returns the <left> and <right> range value of instances in a loop where <instance> is one of those instances. If <instance> is not in a loop, both <left> and <right> are assigned -1.

7.4.60. gns_GetCorrespondingInstance

```
CorrespondingInstance *gns_GetCorrespondingInstance(char *name)
```

Returns the instance in the circuit corresponding to an instance in the model. An instance in a loop can be referenced using the instance name vectorized. <name> can be hierarchical.

7.4.61. gns_GetCorrespondingInstanceConnectorSignal

```
Signal *gns_GetCorrespondingInstanceConnectorSignal(CorrespondingInstance
*ins, char *name)
```

Returns the signal connected to the connector <name> of <instance>.

7.4.62. gns_GetCorrespondingInstanceName

```
char *gns_GetCorrespondingInstanceName(CorrespondingInstance *crt)
```

Returns the name given by YagleGNS to a recognized instance.

7.4.63. gns_GetGeneric

```
int gns_GetGeneric(char *name)
```

Returns the integer value of a generic variable in the current instance.

7.4.64. gns_GetCurrentArchi

```
char *gns_GetCurrentArchi()
```

Returns the architecture name of the current instance.

7.4.65. gns_GetCurrentModel

```
char *gns_GetCurrentModel()
```

Returns the model name of the current instance.

7.4.66. gns_GetCurrentInstance

```
char *gns_GetCurrentInstance()
```

Returns the name of the current instance.

7.4.67. callfunc

```
void *callfunc(char *funcname, ... )
```

Generates a function call. The number of arguments is variable. When the results of 'callfunc' is used, <funcname> will be called with the given arguments.

7.4.68. gns_DriveSpiceNetlistGroup

```
void gns_DriveSpiceNetlistGroup(List *list, char *filename)
```

Saves the <list> of netlist in <filename> using the spice format.

7.4.69. gns_AddCapa

```
void gns_AddCapa(Netlist *fig, char *con_name, double capa)
```

Add <capa> between <con_name> member of <netlist> and the ground.

7.4.70. gns_AddResi

```
void gns_AddResi(Netlist *fig, char *con1_name, char *con2_name, double
resi)
```

Add <resi> between <con1_name> and <con2_name> member of <netlist>.

7.4.71. gns_AddLineRC

```
void gns_AddLineRC(Netlist *fig, char *con1_name, char *con2_name, double
resi, double capa1, double capa2)
```

Add <resi> between <con1_name> and <con2_name> member of <netlist>. Add <capa(n)> to <con(n)_name>

7.4.72. gns_RunGNS

```
GNSRun *gns_RunGNS(Netlist *netlist, char *celldir, char *libname)
```

Initiate a GNS recognition on the specified netlist. <dir> is the directory where to search for GNS rules and actions. <lib> is the library file

describing the rules and actions to use. If they are set to NULL, befault directory and library file will be used.

7.4.73. gns_DestroyGNSRun

```
void gns_DestroyGNSRun(GNSRun *afg)
```

Destroys the <gnsrun>.

7.4.74. gns_EnterGNSContext

```
void gns_EnterGNSContext(GNSRun *gnsrun, char *instance)
```

Changes GNS environment to match the one of <instance_name> in the <gnsrun>. The old environment is pushed into a stack. It can be retreive (poped) using gns_ExitGNSContext().

7.4.75. gns_ExitGNSContext

```
void gns_ExitGNSContext()
```

Changes GNS environment to the previous one.

7.4.76. gns_GetBlackboxNetlist

```
Netlist *gns_GetBlackboxNetlist(char *name)
```

Returns the Netlist <name>. The Netlist is searched in the main netlist.

7.4.77. gns_IsTopLevel

```
int gns_IsTopLevel()
```

Indicates if the current instance if a top level of genius recognition.

7.4.78. gns_RenameInstanceFigure

```
void gns_RenameInstanceFigure(Netlist *lf, char *instance, char
*origname, char *newname)
```

Changes the figure name of <instance> in <netlist>. The <original figure name> is replaced by <new figure name>. If instance is NULL, all instances will be checked for a figure name change.

7.4.79. gns_FillBlackBoxes

```
void gns_FillBlackBoxes(Netlist *lf, List *morenetlist)
```

Will try to retreive the blackbox figures from the list of netlist <modellist> then in the original netlist, flatten the <netlist> blackbox instances to transistor level. The <modellist> is not freed.

7.4.80. gns_ChangeNetlistName

```
void gns_ChangeNetlistName(Netlist *lf, char *name)
```

Changes the name of the netlist <netlist> by <name>.

7.4.81. gns_GetGNSTopLevels

```
List *gns_GetGNSTopLevels(GNSRun *gnsrun)
```

Returns the list of all the instances at the top level of the <gnsrun>. Each element of the list is a string containing an instance name.

7.4.82. gns_CutNetlist

```
Netlist *gns_CutNetlist(GNSRun *gnsrun)
```

Returns a new netlist where the instances recognized at top level of <gnsrun> are cut. For each of those instances, a new figure at transistor level is also created and instanciated in the returned netlist. Concerning the parasitics, all couplings between blocks are put to ground and depending on the connector direction or user which, the RC trees are put inside or outside the instances.

7.4.83. gns_ShowOutsideInfo

```
void gns_ShowOutsideInfo(char *signame, FILE *f)
```

Drives the connections of signal <signal> outside of the GNS instance. The result is put in file <file>.

7.4.84. gns_REJECT_INSTANCE

```
void gns_REJECT_INSTANCE()
```

If called will result in the exclusion of the current instance from the instances to keep at the top level. This option overrides the GNS LIBRARY file settings.

7.4.85. gns_KEEP_INSTANCE

```
void gns_KEEP_INSTANCE()
```

If called will result in the inclusion of the current instance in the instances to keep at the top level. This option overrides the GNS LIBRARY file settings.

7.4.86. gns_REJECT_MODEL

```
void gns_REJECT_MODEL()
```

If called will result in the exclusion of all the instances of the current instance model from the instances to keep at the top level. This option overrides the GNS LIBRARY file settings.

7.4.87. gns_KEEP_MODEL

```
void gns_KEEP_MODEL()
```

If called then all the instances of the current instance model are kept at the top level. This option overrides the GNS LIBRARY file settings.

7.4.88. gns_GetWorkingFigureName

```
char *gns_GetWorkingFigureName()
```

Returns the name of the figure the GNS recognition is working on.

7.4.89. gns_IsVss

```
int gns_IsVss(Signal *sig)
```

Returns 1 if the signal <sig> is a vss alim signal, 0 otherwise.

7.4.90. gns_IsVdd

```
int gns_IsVdd(Signal *sig)
```

Returns 1 if the signal <sig> is a vdd alim signal, 0 otherwise.

7.4.91. gns_IsBlackBox

```
int gns_IsBlackBox()
```

Returns 1 if the netlist corresponding with the current instance is a blackbox, 0 otherwise.

7.4.92. gns_GetSignalVoltage

```
double gns_GetSignalVoltage(char *name)
```

Returns the value of the voltage source set on the signal `name`. `name` *is by delault a genius model signal name but if prefixed with 'ext:' the signal is considered to be in the original netlist. If no voltage has been set on the signal, value -10000 is returned.*

7.4.93. gns_GetSignalVoltageSwing

```
int gns_GetSignalVoltageSwing(char *name, double *low, double *high)
```

Gives the voltage swing of the signal `name`. `low` *is the lowest voltage possible on the signal and* `high` *the highest.* `name` *is must genius model signal name.*

7.5. Utility

The MBK API provides a set of utility functions to allow creation and manipulation of chain list and hash table objects.

7.5.1. fopen

```
FILE *fopen(char *name, char *mode)
```

Opens a file in the specified mode and returns a pointer

`name`	Name of the file to be opened
`mode`	Available modes are `r` for reading, `w` for writing and `a` for appending
EXAMPLE	`set file [fopen "design.stat" w]`

7.5.2. fclose

```
int fclose(FILE *f)
```

closes a file

`f`	Pointer on the file to be closed
EXAMPLE	`fclose $file`

7.5.3. mbk_Sort

```
void mbk_Sort(int *index_array, void **value_array, int nbelem)
```

Sorts the array of values 'value_array' according to the indexes stored in the array 'index_array'. 'nbelem' is number of elements of both the arrays 'index_array' and 'value_array'.

7.5.4. mbk_FreeList

```
void mbk_FreeList(List *lst)
```

Frees a List but not the items of the list.

7.5.5. mbk_GetListItem

```
void *mbk_GetListItem(List *lst)
```

Returns the current list item. The item is a pointer on void that must be casted to the desired type.

7.5.6. mbk_AddListItem

```
List *mbk_AddListItem(List *lst, void *item)
```

Adds an item at the begining of a list. The new list head is returned.

7.5.7. mbk_AppendList

```
List *mbk_AppendList(List *lst1, List *lst2)
```

Appends list <l2> at the end of list <l1>. The new list head is returned. NB: <l2> should not be used anymore.

7.5.8. mbk_GetListNext

```
List *mbk_GetListNext(List *lst)
```

Returns the next node of the list.

7.5.9. mbk_EndofList

```
int mbk_EndofList(List *lst)
```

Returns 0 if lst is not the end of the list else a value different from 0. The end of the list can also be tested with (lst == NULL).

7.5.10. mbk_NewHashTable

```
HashTable *mbk_NewHashTable(int size)
```

Creates a hash table with the initial size 'size'. However, if the hash table becomes too crowded, it is automatically resized.

7.5.11. mbk_FreeHashTable

```
void mbk_FreeHashTable(HashTable *htable)
```

Deletes the hash table 'htable'.

7.5.12. mbk_AddHashItem

```
long mbk_AddHashItem(HashTable *htable, void *key, long value)
```

Stores the element 'value' in the hash table 'htable', according to the key 'key'.

7.5.13. mbk_GetHashItem

```
long mbk_GetHashItem(HashTable *htable, void *key)
```

Retrieves the element stored according to the key 'key' in the hash table 'htable'

7.5.14. mbk_IsEmptyHashItem

```
int mbk_IsEmptyHashItem(long value)
```

Tests the returned value of the functions mbk_AddHashItem and mbk_GetHashItem.

7.6. Database

The database API provides a set of functions to create and manage internal databases. This can be useful as a means of manipulating complex data structures within GNS actions.

7.6.1. dtb_Load

```
int dtb_Load(char *name)
```

dtb_Load(<database>) creates a new data base named <database>. If the database was already created, it will be cleaned.

7.6.2. dtb_Save

```
void dtb_Save(char *name)
```

dtb_Save(<database>) saves the <database> to a file. The file name will be .<database>.dtb and can be found in the current directory.

7.6.3. dtb_Clean

```
void dtb_Clean(char *name)
```

dtb_Clean(<database>) removes of entries in the <database>.

7.6.4. dtb_SetChar

```
void dtb_SetChar(char *dbtname, char *name, char value)
```

dtb_SetChar(<database>, <varname>, <value>) creates or updates the entry <varname> in the <database> with the value <value> expressed as a character. In case of type mismatch, the old type will be overriden.

7.6.5. dtb_SetString

```
void dtb_SetString(char *dbtname, char *name, char *value)
```

dtb_SetString(<database>, <varname>, <value>) creates or updates the entry <varname> in the <database> with the value <value> expressed as a string. In case of type mismatch, the old type will be overriden.

7.6.6. dtb_SetLong

```
void dtb_SetLong(char *dbtname, char *name, long value)
```

dtb_SetLong(<database>, <varname>, <value>) creates or updates the entry <varname> in the <database> with the value <value> expressed as a long integer. In case of type mismatch, the old type will be overriden.

7.6.7. dtb_SetInt

```
void dtb_SetInt(char *dbtname, char *name, int value)
```

dtb_SetInt(<database>, <varname>, <value>) creates or updates the entry <varname> in the <database> with the value <value> expressed as an integer. In case of type mismatch, the old type will be overriden.

7.6.8. dtb_SetDouble

```
void dtb_SetDouble(char *dtbname, char *name, double value)
```

dtb_SetDouble(<database>, <varname>, <value>) creates or updates the entry <varname> in the <database> with the value <value> expressed as a double. In case of type mismatch, the old type will be overriden.

7.6.9. dtb_GetDouble

```
double dtb_GetDouble(char *dtbname, char *name)
```

dtb_GetDouble(<database>, <varname>) returns the double value of the entry <varname> in the <database>. If the entry does not exist or the type mismatchs, 0.0 is returned.

7.6.10. dtb_GetInt

```
int dtb_GetInt(char *dtbname, char *name)
```

dtb_GetInt(<database>, <varname>) returns the integer value of the entry <varname> in the <database>. If the entry does not exist or the type mismatchs, 0 is returned.

7.6.11. dtb_GetLong

```
long dtb_GetLong(char *dtbname, char *name)
```

dtb_GetLong(<database>, <varname>) returns the long integer value of the entry <varname> in the <database>. If the entry does not exist or the type mismatchs, 0 is returned.

7.6.12. dtb_GetString

```
char *dtb_GetString(char *dtbname, char *name)
```

dtb_GetString(<database>, <varname>) returns the string value of the entry <varname> in the <database>. If the entry does not exist or the type mismatchs, NULL is returned.

7.6.13. dtb_GetChar

```
char dtb_GetChar(char *dtbname, char *name)
```

dtb_Getchar(<database>, <varname>) returns the character value of the entry <varname> in the <database>. If the entry does not exist or the type mismatchs, ' ' is returned.

7.6.14. dtb_RemoveEntry

```
void dtb_RemoveEntry(char *dtbname, char *name)
```

dtb_RemoveEntry(<database>, <varname>) removes the entry <varname> from the <database>. If the entry does not exist, nothing is done.

7.6.15. dtb_Create

```
void dtb_Create(char *name)
```

dtb_Create(<database>) creates a new database named <database>. If the database already exits, the function call has no effects.

7.7. SPICE Simulation

The SIM API provides a set of functions to perform and take measures from SPICE simulations of internally generated netlists using any external SPICE simulator. Functions are provided to position input waveforms, set simulation parameters, and take measurements of complex properties such as peak noise and setup/hold constraints.

7.7.1. sim_SetSimulatorType

```
void sim_SetSimulatorType(SimulationContext *sc, char *type)
```

sim_SetSimulatorType specifies the type of simulator towards whom the simulation netlists are created. Supported simulators types are 'NGSPICE' and 'ELDO'.

7.7.2. sim_CreateContext

```
SimulationContext *sim_CreateContext(Netlist *netlist)
```

sim_FreeContext deletes a previously created simulation context. The netlist associated with the simulation context is not affected.

7.7.3. sim_CreateNetlistContext

```
SimulationContext *sim_CreateNetlistContext()
```

sim_CreateNetlistContext creates a simulation context, relatively to the current gns netlist which is flattened to transistor level.

7.7.4. sim_GetContextNetlist

```
Netlist *sim_GetContextNetlist(SimulationContext *sc)
```

sim_GetContextNetlist returns the netlist the simulation context `sc` *is associated to.*

7.7.5. sim_SetDelayVTH

```
void sim_SetDelayVTH(SimulationContext *sc, double vth)
```

sim_SetDelayVTH sets the threshold voltage for delay calculations. Delay is computed between the vth-crossing ins

---

7.7.6. sim_SetSimulationSlope

```
void sim_SetSimulationSlope(SimulationContext *sc, double slope)
```

sim_SetSimulationSlope sets the default input slope for electrical simulation. The unit is the second.

7.7.7. sim_SetSimulationTime

```
void sim_SetSimulationTime(SimulationContext *sc, double time)
```

sim_SetSimulationTime sets the duration of the electrical simulation. Unit is SECOND.

7.7.8. sim_SetSimulationStep

```
void sim_SetSimulationStep(SimulationContext *sc, double step)
```

sim_SetSimulationStep sets the step used during the electrical simulation. The unit is the second.

7.7.9. sim_SetSimulationSupply

```
void sim_SetSimulationSupply(SimulationContext *sc, double v_max)
```

sim_SetSimulationSupply sets the applied power supply during the electrical simulation. The unit is the volt.

7.7.10. sim_SetInputSwing

```
void sim_SetInputSwing(SimulationContext *sc, double v_vss, double v_max)
```

sets the swing to use for input connector/signal. Those values are used for instance when setting an input slope on a connector.

7.7.11. sim_SetOutputSwing

```
void sim_SetOutputSwing(SimulationContext *sc, double v_vss, double
v_max)
```

sets the swing to use for output signal when computing a delay to the output signal or the slope on an output signal.

7.7.12. sim_GetSimulationSupply

```
double sim_GetSimulationSupply()
```

sim_GetSimulationSupply returns the simulation voltage defined for the alimentation connector. The unit is the volt.

7.7.13. sim_AddSimulationTechnoFile

```
void sim_AddSimulationTechnoFile(SimulationContext *sc, char *tech_file)
```

sim_AddSimulationTechnoFile adds a technology file in list of technology files used to parametrize the electrical simulation.

7.7.14. sim_SetSimulationCall

```
void sim_SetSimulationCall(SimulationContext *sc, char *sim_call)
```

sim_SetSimulationCall sets the string which will be called to run the electrical simulator.

7.7.15. sim_NoiseSetAnalyseType

```
void sim_NoiseSetAnalyseType(SimulationContext *sc, char noise_type)
```

sim_NoiseSetAnalyseType sets the type of noise analysis. Allowed values are SIM_MIN and SIM_MAX.

7.7.16. sim_SetSimulationOutputFile

```
void sim_SetSimulationOutputFile(SimulationContext *sc, char
*output_file)
```

sim_SetSimulationOutputFile specifies the extension of the file generated by the electrical simulator.

7.7.17. sim_AddStuckLevel

```
void sim_AddStuckLevel(SimulationContext *sc, char *node, int level)
```

sim_AddStuckLevel stucks the node to VDD if level is 1, to GND if level is 0.

7.7.18. sim_AddStuckLevelVector

```
void sim_AddStuckLevelVector(SimulationContext *sc, char *node, char
*level)
```

sim_AddStuckLevelVector stucks the input bit vector to the hexadecimal value (VDD if the bit value is 1, GND if the bit value is 0).

7.7.19. sim_AddStuckVoltage

```
void sim_AddStuckVoltage(SimulationContext *sc, char *node, double
voltage)
```

sim_AddStuckVoltage stucks the input to the value.

7.7.20. sim_AddSlope

```
void sim_AddSlope(SimulationContext *sc, char *node, double start_time,
double transition_time, char sense)
```

sim_AddSlope sets a rising slope on the input (an internal signal) if sense is 'U', a falling slope if sense is 'D'.

7.7.21. sim_SetExternalCapacitance

```
void sim_SetExternalCapacitance(SimulationContext *sc, char *node, double
value)
```

sim_SetExternalCapacitance sets a capacitance `value` *on the toplevel connector* `node`*.*

7.7.22. sim_AddWaveForm

```
void sim_AddWaveForm(SimulationContext *sc, char *node, double trise,
double tfall, double periode, char *pattern)
```

sim_AddWaveForm sets rising and falling transitions on the node, according to the string 'pattern'. Specifying 1 in 'pattern' sets rising slope, 0 a falling slope.

7.7.23. sim_AddInitLevel

```
void sim_AddInitLevel(SimulationContext *sc, char *node, int level)
```

sim_AddInitLevel initialize a node voltage to VDD if level = 1, to GND if level = 0.

7.7.24. sim_AddInitVoltage

```
void sim_AddInitVoltage(SimulationContext *sc, char *node, double
voltage)
```

sim_AddInitVoltage initialize a node voltage to the value voltage.

7.7.25. sim_AddOutLoad

```
void sim_AddOutLoad(SimulationContext *sc, char *node, double load)
```

sim_AddOutLoad adds the capacitance 'load' on the specified 'node' output connector.

7.7.26. sim_AddMeasure

```
void sim_AddMeasure(SimulationContext *sc, char *node)
```

sim_AddMeasure prints the signal voltage to the simulator output file. This function is needed to compute timing.

7.7.27. sim_AddMeasureCurrent

```
void sim_AddMeasureCurrent(SimulationContext *sc, char *node)
```

sim_AddMeasureCurrent prints the node current to the simulator output file.

7.7.28. sim_RunSimulation

```
void sim_RunSimulation(SimulationContext *sc, char *sim_call)
```

sim_RunSimulation launches the electrical simulation.

7.7.29. sim_ExtractMinSlope

```
double sim_ExtractMinSlope(SimulationContext *sc, char *node)
```

After a simulation run, sim_ExtractMinSlope extracts the minimum slope of a node.

7.7.30. sim_ExtractMaxSlope

```
double sim_ExtractMaxSlope(SimulationContext *sc, char *node)
```

After a simulation run, sim_ExtractMaxSlope extracts the minimum slope of a node.

7.7.31. sim_ExtractMinDelay

```
double sim_ExtractMinDelay(SimulationContext *sc, char *node_a, char
*node_b)
```

After a simulation run, sim_ExtractMinDelay extracts the minimum delay between two nodes.

7.7.32. sim_ExtractMaxDelay

```
double sim_ExtractMaxDelay(SimulationContext *sc, char *node_a, char
*node_b)
```

After a simulation run, sim_ExtractMaxDelay gets the maximum delay between two nodes.

7.7.33. sim_ExtractMinTransitionDelay

```
double sim_ExtractMinTransitionDelay(SimulationContext *sc, char *node_a,
char *node_b, char *transition)
```

After a simulation run, sim_ExtractMinTransitionDelay extracts the mininum delay between two nodes, see sim_ExtractMinDelay. Parameter 'transition' can be for example "U1D2". In this configuration, delay will be extracted between the second rise transition of 'node_a' and the third falling transition of 'node_b'.

7.7.34. sim_ExtractMaxTransitionDelay

```
double sim_ExtractMaxTransitionDelay(SimulationContext *sc, char *node_a,
char *node_b, char *transition)
```

After a simulation run, sim_ExtractMaxTransitionDelay extracts the maximum delay between two nodes, see sim_ExtractMaxDelay. Parameter 'transition' can be for example "U1D2". In this configuration, delay will be extracted between the second rise transition of 'node_a' and the third falling transition of 'node_b'.

7.7.35. sim_ExtractMinTransitionSlope

```
double sim_ExtractMinTransitionSlope(SimulationContext *sc, char *node,
char *transition)
```

After a simulation run, sim_ExtractMinTransitionSlope extracts the minimum slope of a node. Parameter 'transition' can be for example "U1". In this configuration, slope will be extracted from the second rising transition of the node.

7.7.36. sim_ExtractMaxTransitionSlope

```
double sim_ExtractMaxTransitionSlope(SimulationContext *sc, char *node,
char *transition)
```

After a simulation run, sim_ExtractMaxTransitionSlope extracts the maximum slope of a node. Parameter 'transition' can be for example "D2". In this configuration, slope will be extracted from the third falling transition of the node.

7.7.37. sim_ComputeSetup

```
double sim_ComputeSetup(SimulationContext *sc, char *data, double
tstart_d, double tslope_d, char sense_d, char *cmd, double t_start_min_c,
double t_start_max_c, double tslope_c, char sense_c, char *mem, int
data_val)
```

sim_ComputeSetup computes the setup time of 'data' relatively to 'command'. Setup time is computed by observing the latest 'data' transition, relatively to 'command', that generates a transition on the memory point. data : name of the data tstart_d : start time of the pulse on data tslope_d : slope of the pulse on data sens_d : transition of the data : 'U' for rising, 'D' for falling cmd : name of the command tstart_min : minimum starting time of the pulse on cmd tstart_max : maximum starting time of the pulse on cmd tslope_c : slope of the pulse on cmd sens_c : transition of the cmd : 'U' for rising, 'D' for falling mem : name of the memory point data_val : expected value on mem : 1 for VDD, 0 for VSS

7.7.38. sim_ComputeHold

```
double sim_ComputeHold(SimulationContext *sc, char *data, double
tstart_d, double tslope_d, char sense_d, char *cmd, double t_start_min_c,
double t_start_max_c, double tslope_c, char sense_c, char *mem, int
data_val)
```

sim_ComputeHold computes the hold time of 'data' relatively to 'command'. Hold time is computed by observing the latest 'data' transition, relatively to command, that doesn't generate any transition on the memory point. data : name of the data tstart_d : start time of the pulse on data tslope_d : slope of the pulse on data sens_d : transition of the data : 'U' for rising, 'D' for falling cmd : name of the command tstart_min : minimum starting time of the pulse on cmd tstart_max : maximum starting time of the pulse on cmd tslope_c : slope of the pulse on cmd sens_c : transition

of the cmd : 'U' for rising, 'D' for falling mem : name of the memory point data_val : expected value on mem : 1 for VDD, 0 for VSS

7.7.39. sim_ComputeAccess

```
double sim_ComputeAccess(SimulationContext *sc, char *dout, int dout_val,
char *cmd, double tstart_c, double tslope_c, char sens_c, char *mem, int
mem_val, double *out_slope)
```

sim_ComputeAccess gives the access time of 'dout' relatively to 'command'. Access time is computed by observing the delay between the transition on 'command' and the transition on 'dout'.

7.7.40. elp_GetCapaFromConnector

```
double elp_GetCapaFromConnector(SimulationContext *sc, Connector *locon)
```

elp_GetCapaFromConnector gives the capacitance of a transistor's connector.

7.7.41. sim_ComputeDelay

```
List *sim_ComputeDelay(SimulationContext *sc, char *input, char sens,
List *list_output)
```

sim_ComputeDelay gives a list of delays between a constant input and a list of outputs.

7.7.42. sim_ComputeMaxDelayTransition

```
double sim_ComputeMaxDelayTransition(SimulationContext *sc, char *input,
double input_start, double input_slope, char *output, char *transition)
```

sim_ComputeMaxDelayTransition gives the maximum delay corresponding to a transition between the input and the output.

7.7.43. sim_ComputeMinDelayTransition

```
double sim_ComputeMinDelayTransition(SimulationContext *sc, char *input,
double input_start, double input_slope, char *output, char *transition)
```

sim_ComputeMinDelayTransition gives the minimum delay corresponding to a transition between the input and the output.

7.7.44. sim_GetTimingFromList

```
Timing *sim_GetTimingFromList(List *list)
```

sim_GetTimingFromList gives a timing object corresponding to a list.

7.7.45. sim_GetTimingNext

```
Timing *sim_GetTimingNext(Timing *timing)
```

sim_GetTimingNext gives the next timing object.

7.7.46. sim_GetTiming

```
Timing *sim_GetTiming(char *root, char *node)
```

sim_GetTiming retrieves the timing between the root node name 'rootname' and the destination node name 'nodename'.

7.7.47. sim_GetTimingByEvent

```
Timing *sim_GetTimingByEvent(char *root, char *node, char *event)
```

sim_GetTimingByEvent retrieves the timing between the root node name 'rootname' and the destination node name 'nodename'. This timing must respect the good event on 'rootname' and 'nodename'. Event is the expected event from root to node, it can be 'U' (rising) or 'D' (falling) and can be followed by an integer.

7.7.48. sim_GetTimingDelay

```
double sim_GetTimingDelay(Timing *timing)
```

sim_GetTimingDelay gets the delay corresponding to the timing.

7.7.49. sim_GetTimingMinDelay

```
double sim_GetTimingMinDelay(Timing *timing)
```

sim_GetTimingMinDelay gets the minimun delay corresponding to the timing.

7.7.50. sim_GetTimingMaxDelay

```
double sim_GetTimingMaxDelay(Timing *timing)
```

sim_GetTimingMaxDelay gets the maximun delay corresponding to the timing.

7.7.51. sim_GetTimingSlope

```
double sim_GetTimingSlope(Timing *timing)
```

sim_GetTimingSlope gets the slope corresponding to the timing.

7.7.52. sim_GetTimingMinSlope

```
double sim_GetTimingMinSlope(Timing *timing)
```

sim_GetTimingMinSlope gets the minimum slope corresponding to the timing.

7.7.53. sim_GetTimingMaxSlope

```
double sim_GetTimingMaxSlope(Timing *timing)
```

sim_GetTimingMaxSlope gets the maximum slope corresponding to the timing.

7.7.54. sim_GetTimingRoot

```
char *sim_GetTimingRoot(Timing *timing)
```

sim_GetTimingRoot gets the name of the root node corresponding to the timing.

7.7.55. sim_GetTimingNode

```
char *sim_GetTimingNode(Timing *timing)
```

sim_GetTimingNode gets the name of the node (destination) corresponding to the timing.

7.7.56. sim_GetTimingRootInNetlist

```
char *sim_GetTimingRootInNetlist(Timing *timing)
```

sim_GetTimingRootInNetlist gets the name of the root node in the netlist corresponding to the timing.

7.7.57. sim_GetTimingNodeInNetlist

```
char *sim_GetTimingNodeInNetlist(Timing *timing)
```

sim_GetTimingNodeInNetlist gets the name of node (destination) in the netlist corresponding to the timing.

7.7.58. sim_GetTimingRootEvent

```
char sim_GetTimingRootEvent(Timing *timing)
```

sim_GetTimingRootEvent gets the event of the root node corresponding to the timing. Event is SIM_FALL or SIM_RISE.

7.7.59. sim_GetTimingNodeEvent

```
char sim_GetTimingNodeEvent(Timing *timing)
```

sim_GetTimingNodeEvent gets the event of the node (destination) corresponding to the timing. Event is SIM_FALL or SIM_RISE.

7.7.60. sim_NoiseExtract

```
void sim_NoiseExtract(SimulationContext *sc, char *node, double vthnoise,
double tinit, double tfinal)
```

After a simulation run, sim_NoiseExtract extracts the maximum noise on a node between two moments. The initial time (tinit) and the tfinal time (tfinal) represent the timing bounds to extract noise. vthnoise is the threshold voltage to extract noise (percentage of vdd).

7.7.61. sim_NoiseGetVth

```
double sim_NoiseGetVth(SimulationContext *sc, char *name)
```

sim_NoiseGetVth gets the noise threshold voltage on the node 'name'.

7.7.62. sim_NoiseGetPeakList

```
List *sim_NoiseGetPeakList(SimulationContext *sc, char *name)
```

sim_NoiseGetPeakList gets a list of peaks relatively to node 'name'.

7.7.63. sim_NoiseGetMomentList

```
List *sim_NoiseGetMomentList(SimulationContext *sc, char *name)
```

sim_NoiseGetMomentList gets a list of moment of passage on noise threshold voltage relatively to node 'name'.

7.7.64. sim_NoiseGetMoment

```
double sim_NoiseGetMoment(SimulationContext *sc, List *noise_tclist)
```

sim_NoiseGetMoment gets a the moment of passage on noise threshold voltage from a pointer on a pointer list returned by sim_NoiseGetMomentList.

7.7.65. sim_NoiseGetPeakValue

```
double sim_NoiseGetPeakValue(SimulationContext *sc, char *name, List
*noise_peaklist)
```

sim_NoiseGetPeakValue gets the peak value (voltage) relatively to the node 'name' and a pointer on a List returned by sim_NoiseGetPeakList.

7.7.66. sim_NoiseGetPeakMoment

```
double sim_NoiseGetPeakMoment(SimulationContext *sc, List
*noise_peaklist)
```

sim_NoiseGetPeakMoment gets the peak moment relatively to a pointer on a List returned by sim_NoiseGetPeakList.

7.7.67. sim_NoiseExtractMaxPeakValue

```
double sim_NoiseExtractMaxPeakValue(SimulationContext *sc, char *name)
```

sim_NoiseExtractMaxPeakValue gets the maximum peak value relatively to the node 'name'.

7.7.68. sim_NoiseExtractMinPeakValue

```
double sim_NoiseExtractMinPeakValue(SimulationContext *sc, char *name)
```

sim_NoiseExtractMinPeakValue gets the minimum peak value relatively to the node 'name'.

7.7.69. sim_NoiseExtractMaxPeakMoment

```
double sim_NoiseExtractMaxPeakMoment(SimulationContext *sc, char *name)
```

sim_NoiseExtractMaxPeakMoment gets the moment of the maximum peak relatively to the node 'name'.

7.7.70. sim_NoiseExtractMinPeakMoment

```
double sim_NoiseExtractMinPeakMoment(SimulationContext *sc, char *name)
```

sim_NoiseExtractMinPeakMoment gets the moment of the minimum peak relatively to the node 'name'.

7.7.71. sim_NoiseGetMomentBeforePeak

```
double sim_NoiseGetMomentBeforePeak(SimulationContext *sc, char *name,
List *peak)
```

sim_NoiseGetMomentBeforePeak gets the moment of passage on noise threshold voltage relatively to node 'name' and before a peak which is a pointer on a List returned by sim_NoiseGetPeakList.

7.7.72. sim_NoiseGetMomentAfterPeak

```
double sim_NoiseGetMomentAfterPeak(SimulationContext *sc, char *name,
List *peak)
```

sim_NoiseGetMomentAfterPeak gets the moment of passage on noise threshold voltage relatively to node 'name' and after a peak which is a pointer on a List returned by sim_NoiseGetPeakList.

7.7.73. sim_DriveNodeState

```
void sim_DriveNodeState(SimulationContext *sc, char *filename, char
*node_ref, char *node_state2drive, char type)
```

sim_DriveNodeState drives successives states of the 'node_state2drive' in the file 'filename' (which also contains file's extension). 'node_ref' is the node reference, 'type' can be SIM_RISE or SIM_FALL and represents the event on 'node_ref' which will sample 'node_state2drive'.

7.7.74. sim_ExtractCommutInstant

```
double sim_ExtractCommutInstant(SimulationContext *sc, char *node, double
voltage)
```

sim_ExtractCommutInstant extract the first instant when the node reach the voltage value 'voltage'.

7.7.75. sim_DriveTransistorAsInstance

```
void sim_DriveTransistorAsInstance(SimulationContext *sc, char mode)
```

sim_DriveTransistorAsInstance(<context>, <mode>) indicates is the transistor should be driven as instances. eg. M124 src grid drain bulk ... => XM124 src grid drain bulk ... the value for mode is 'y' to enable the transformation else 'n'

7.7.76. sim_AddSpiceMeasure

```
void sim_AddSpiceMeasure(SimulationContext *sc, char *delay, char *slope,
char *sig1, char *sig2, char *transition, char delay_type)
```

7.7.77. sim_AddSpiceMeasureSlope

```
void sim_AddSpiceMeasureSlope(SimulationContext *sc, char *slope, char
*sig, char *transition, char delay_type)
```

Add a slope measure of sig that can be extracted by the label (slope). Transition is a string containing the transition of sig. Tolerated transition are 'U' and 'D'. The <delay_type> can be SIM_MIN or SIM_MAX meaning the node choosen to compute the delay or the slope is the closer or the farther one.

7.7.78. sim_AddSpiceMeasureDelay

```
void sim_AddSpiceMeasureDelay(SimulationContext *sc, char *delay, char
*sig1, char *sig2, char *transition, char delay_type)
```

Add a measure of delay between sig1 and sig2 that can be extracted by the label (delay). Transition is a string containing the transition of sig1 and sig2. The <delay_type> can be SIM_MIN or SIM_MAX meaning the node choosen to compute the delay or the slope is the closer or the farther one. Tolerated transition are 'U' and 'D'.

7.7.79. sim_ReadMeasure

```
double sim_ReadMeasure(char *filename, char *label)
```

Reads the simulation results from `filename` *and return the value corresponding to the measure* `label`*. On failure, returns* `-1`*.*

7.7.80. sim_ResetMeasures

```
void sim_ResetMeasures(SimulationContext *model)
```

Resets all the measures set in the simulation context.

7.7.81. sim_GetSpiceMeasureSlope

```
double sim_GetSpiceMeasureSlope(SimulationContext *model, char *label)
```

Returns the slope computed for the label, 0.0 if slope has not been computed.

7.7.82. sim_GetSpiceMeasureDelay

```
double sim_GetSpiceMeasureDelay(SimulationContext *model, char *label)
```

Returns the delay computed for the label, 0.0 if delay has not been computed.

7.7.83. sim_SpiceMeasure

```
void sim_SpiceMeasure(SimulationContext *model, char *delay, double
*valued, char *slope, double *values, char *sig1, char *sig2, char
*transition, char delay_type)
```

Add a delay measure between sig1 and sig2 that can be extracted by the label (delay). Add a slope measure of sig that can be extracted by the label (slope). Transition is a string containing the transition of sig1 and sig2. Tolerated transition are 'U' and 'D'. The <delay_type> can be SIM_MIN or SIM_MAX meaning the node choosen to compute the delay or the slope is the closer or the farther one. After simulation completed get the delay and the slope computed for each label and store it into the adress pointed by valued(delay) and values(slope).

7.7.84. sim_SpiceMeasureDelay

```
void sim_SpiceMeasureDelay(SimulationContext *model, char *delay, double
*value, char *sig1, char *sig2, char *transition, char delay_type)
```

Add a measure of delay between sig1 and sig2 that can be extracted by the label (delay). Transition is a string containing the transition of sig1 and sig2. Tolerated transition are 'U' and 'D'. The <delay_type> can be SIM_MIN or SIM_MAX meaning the node choosen to compute the delay or the slope is the closer or the farther one. After simulation completed get the delay computed for the label and store it into the adress pointed by value.

7.7.85. sim_SpiceMeasureSlope

```
void sim_SpiceMeasureSlope(SimulationContext *model, char *slope, double
*value, char *sig, char *transition, char delay_type)
```

Add a slope measure of sig that can be extracted by the label (slope). Transition is a string containing the transition of sig. Tolerated transition are 'U' and 'D'. The <delay_type> can be SIM_MIN or SIM_MAX meaning the node choosen to compute the delay or the slope is the closer or the

farther one. After simulation completed get the slope computed for the label and store it into the adress pointed by value.

7.7.86. sim_DefineInclude

```
void sim_DefineInclude(SimulationContext *sc, char *filename)
```

Sets the filename containing the netlist to apply the pattern to. When this option is used, the gns rule netlist won't be used for the simulation. It will be replaced by the external file given in <filename> at the simulation time.

7.8. Behavior Generation

The BEG API provides a set of functions to allow easy generation of behavioral models. These internally compiled models can be automatically compressed using loops and vectors and then used to generate either VHDL or Verilog descriptions.

7.8.1. begCreateModel

```
void begCreateModel(char *name)
```

Initializes a behavioral model with the given name making it the current model.

7.8.2. begCreatePort

```
void begCreatePort(char *name, char direction)
```

Adds an I/O port to the current model.

7.8.3. begCreateModelFromConnectors

```
void begCreateModelFromConnectors(char *name, List *connectors)
```

Initializes a behavioral model with the given name and interface, making it the current model.

7.8.4. begCreateModelInterface

```
void begCreateModelInterface(char *name)
```

Initializes a behavioral model with the given name and the physical model interface, making it the current model.

7.8.5. begCreateInterface

```
void begCreateInterface()
```

Initializes a behavioral model and the physical model interface, making it the current model. The model name will be handled by the API.

7.8.6. begRenameSignalsFromModel

```
void begRenameSignalsFromModel()
```

Rename the behavioral model interface with the name of physical connectors. If the corresponding signal is not a connector the behavioral name is prefixed by the behavioral model name.

7.8.7. begAssign

```
begAssign [-weak|-strong] <name> <value> [delay [delayvar]]
```

Creates a simple concurrent assignment in the current model.

7.8.8. begAddBusDriver

```
in C:
begAddBusDriver(char *name, char *condition, char *value, int delay, char
*delayvar)
in TCL:
begAddBusDriver [-normal] [-weak|-strong] [-delays <risedelay>
<falldelay>] <name> <condition> <value> [delay [delayvar]]
```

Adds a driver to a given bussed signal of the current model, creating the signal if necessary.

`-normal`	In verilog, will force the signal to be assigned in a sequential block.
`-weak or -strong`	In verilog, defines the strength of the driver.
`-delays <risedelay> <falldelay>`	Specifies different values for rising and falling. No effect if field <delay> is used.

`name`	Affected signal name.
`value`	Affected expression.
`condition`	Condition for the value to be affected.
`delay`	Delay of the operation. Default is 0ps.
`delayvar`	Delay variable name for defining the delay later.

7.8.9. begAddBusElse

```
begAddBusElse [-normal] [-weak|-strong] [-delays <risedelay> <falldelay>]
<name> <condition> <value> [delay [delayvar]]
```

Adds an else alternative to the previous driver of a given bussed signal of the curent model, creating the signal if necessary.

`-normal`	In verilog, will force the signal to be assigned in a sequential block.
`-weak or -strong`	In verilog, defines the strength of the driver.
`-delays <risedelay> <falldelay>`	Specifies different values for rising and falling. No effect if field <delay> is used.
`name`	Affected signal name.
`value`	Affected expression.
`condition`	Condition for the value to be affected.
`delay`	Delay of the operation. Default is 0ps.
`delayvar`	Delay variable name for defining the delay later.

7.8.10. begAddBusDriverLoop

```
void begAddBusDriverLoop(char *name, char *condition, char *value, char
*loopvar, int delay, char *delayvar)
```

Adds a loop driver to a given bussed signal of the current model, creating the signal if necessary.

7.8.11. begAddBusDriverDoubleLoop

```
void begAddBusDriverDoubleLoop(char *name, char *condition, char *value,
char *loopvar1, char *loopvar2, int delay, char *delayvar)
```

Adds a loop driver to a given bussed signal of the current model, creating the signal if necessary.

7.8.12. begAddMemDriver

```
in C:
begAddMemDriver(char *name, char *condition, char *value, int delay, char
*delayvar)
in TCL:
begAddMemDriver [-normal] [-weak|-strong] [-delays <risedelay>
<falldelay>] <name> <condition> <value> [delay [delayvar]]
```

Adds a driver to a given register signal of the curent model, creating the signal if necessary.

`-normal`	In verilog, will force the signal to be assigned in a sequential block.
`-weak or -strong`	In verilog, defines the strength of the driver.
`-delays <risedelay> <falldelay>`	Specifies different values for rising and falling. No effect if field <delay> is used.
`name`	Affected signal name.
`value`	Affected expression.
`condition`	Condition for the value to be affected.
`delay`	Delay of the operation. Default is 0ps.
`delayvar`	Delay variable name for defining the delay later.

7.8.13. begAddMemDriverLoop

```
void begAddMemDriverLoop(char *name, char *condition, char *value, char
*loopvar, int delay, char *delayvar)
```

Adds a loop driver to a given register signal of the curent model, creating the signal if necessary.

7.8.14. begAddMemDriverDoubleLoop

```
void begAddMemDriverDoubleLoop(char *name, char *condition, char *value,
char *loopvar1, char *loopvar2, int delay, char *delayvar)
```

Adds a loop driver to a given register signal of the curent model, creating the signal if necessary.

7.8.15. begAddMemElse

```
begAddMemElse [-normal] [-weak|-strong] [-delays <risedelay> <falldelay>]
<name> <condition> <value> [delay [delayvar]]
```

Adds an else alternative to the previous driver of a given register signal of the curent model, creating the signal if necessary.

`-normal`	In verilog, will force the signal to be assigned in a sequential block.
`-weak or -strong`	In verilog, defines the strength of the driver.
`-delays <risedelay> <falldelay>`	Specifies different values for rising and falling. No effect if field <delay> is used.
`name`	Affected signal name.
`value`	Affected expression.
`condition`	Condition for the value to be affected.
`delay`	Delay of the operation. Default is 0ps.
`delayvar`	Delay variable name for defining the delay later.

7.8.16. begSaveModel

```
void begSaveModel()
```

Saves the current model to disk.

7.8.17. begKeepModel

```
void begKeepModel()
```

Finalize the current custom-built model.

7.8.18. begDestroyModel

```
void begDestroyModel()
```

Destroy the current model.

7.8.19. begVectorize

```
char *begVectorize(char *radical, int index)
```

Generate a name of type toto(n).

7.8.20. begVarVectorize

```
char *begVarVectorize(char *radical, char *var)
```

Generate a name of type toto(n).

7.8.21. begVectorRange

```
char *begVectorRange(char *radical, int left, int right)
```

Generate a name of type toto(l:r).

7.8.22. begAddWarningCheck

```
void begAddWarningCheck(char *testexpr, char *message)
```

Add a assertion statement which generates a warning on activation.

7.8.23. begAddErrorCheck

```
void begAddErrorCheck(char *testexpr, char *message)
```

Add a assertion statement which generates an error on activation.

7.8.24. begSort

```
void begSort()
```

Sort the drivers in the current behavioural model.

7.8.25. begCompact

```
void begCompact()
```

Compact the current behavioural model by vectorization and loop detection.

7.8.26. begSetDelay

```
void begSetDelay(char *varname, int value)
```

Set the timing delay value associated with a particular delay variable declared by any of the expression creation functions.

7.8.27. begBuildModel

```
void begBuildModel()
```

Automatically create a standard (nom-compacted) behavioral model for a recognized structural model.

7.8.28. begBuildCompactModel

```
void begBuildCompactModel()
```

Automatically create a compact behavioral model for a recognized structural model.

7.8.29. begBiterize

```
void begBiterize()
```

Unvectorise current behavioral model.

7.8.30. begAddSelectDriver

```
void begAddSelectDriver(char *name, char *select, char *when, char
*value, int delay, char *delayvar)
```

Add a with select description. 'select' is the name of the signal selected, 'value' is affected to 'name' when 'when' match to 'select'. When others is represented by 'when' set to string 'default'.

7.8.31. begExport

```
void begExport(char *name)
```

begExport(<name>) creates a copy of the current behavioural figure. The new figure gets the name <name>. This function is useful when it is needed to associate a known user name to a behaviour to easily retreive it knowing it's new name.

7.8.32. begImport

```
void begImport(char *name)
```

begImport(<name>) retreive the behaviour named <name> and merge it into the current behavioural figure.

Chapter 8. Creating a User-Defined Dynamic Library API

8.1. Description

It is an extremely simple task for the user to generate his own APIs which can be dynamically linked into an Avertec verification flow. In order to do this, we provide the genapi tool which, from a set of C source files, together with header files declaring the visible functions, creates a shared library which can be used in exactly the same way as the supplied APIs.

8.2. Executing the Genapi Tool

To generate your own dynamic library for linking with the Avertec tools, you should use the Genapi tool as follows:

```
genapi <f.c g.c ...> -i <interface.h> [-o <lib.so>] [--keep_files] [-kf]
```

This takes the specified set of C source code files, together with a header file declaring the visible functions and generates directly a shared library compatible with GNS actions. The header file must contain full ANSI prototype declarations, since this is used to create wrapper functions for the functions to be made visible. By default the shared library generated (file with ".so" suffix) has the same base name as the header file but this can be modified using the "-o" option. The "-kf" (or "--keep_files") option prevents the removal of the intermediate files generated, such as the source code for the wrapper functions and the Makefile.

Chapter 9. Error Messages

9.1. Warning Messages

Warning: Spice file <name> already loaded
This means that the corresponding transistor-level model has already been loaded, the model retained for recognition is the original model.

Warning: circuit signal <name> matched but at least one external connector in the model is missing in the circuit
This means that the signal matching the connector is connected to no other connector in the circuit. In certain cases, this may be perfectly normal, however, it can mean that something is disconnected in the circuit.

9.2. Fatal Errors

Fatal error while executing program
While excuting the actions, the interpreter can be given instructions causing general protection faults either by manipulating pointers or calling functions. When this happens, this message appears.

variable <name> is not defined in the model
A variable <name> declared in the action function header does not correspond to any generic variable in the model.

forbidden operators 'mod', 'rem', ''**
This means there's a loop statement to compute a generic variable but one of the bounds of the loop contains operator that are not handled by Genius in this case.

put this FOR in another model
This means there is more than one loop in the model that contains a generic variable of unknown value. This is forbidden. The user must use a separate hierarchical level for each loop.

transistor in center of loop forbidden
Transistors can not be handled in a loop. This message is given if a rule flouts this restriction.

instance <insname> already exist in figure <model>
This means that an instance of the model <model> with the name <insname> has already been instantiated. Instance names must be unique within a given model.

discrepancy between figure %s and instance %s in figure %s

This means that a model is instantiated with a number of connectors different from the definition of the model.

no model <name> found

The model <name> is used by a rule but its description cannot be found.

unknown connector (<name>) declared in symmetric connector list

The connector <name> is used in a symmetric connector list but is not defined in the entity of the model.

unknown connector (<name>) declared in coupled connector list

The connector <name> is used in a coupled connector list but is not defined in the entity of the model.

Spice file <name> contain more than one description

A SPICE format transistor-level model file can only contain one SUBCKT corresponding to the model to identify.

Spice file <name> should be a flat transistor netlist

The SPICE format model file <name> must be a flat transistor netlist, this error means that the netlist contains instances.

<num> Out of bounds for signal <signame>

The signal <signame> is declared as a vector but during the recognition, genius tried to access <signal>(<num>) where <num> is not within the signal vector range. Check the model declaration.

connectors <conname1> and <conname2> mismatch in loins <insname1> and <insname2>

When using the instance <insname1> previously recognized, there is a discrepancy between the connector <conname1> and the connector <conname2> of the instance <insname2> instantiated in the model.

No search done on connector '<name1>' signal '<name1>', model '<name1>' must be a connexe graph

This message means that not all of the connectors in the model have been traversed. All the instances must be connected together so that the recognition algorithm can traverse all the instances in the model by jumping between connectors.

connector <model>.<name> is in coupled list but has no symmetry

The connector <name> of the model <model> is used in a coupled connector list but is not a symmetric connector.

no symmetry found for connector <name> in coupled list

The connector <name> of the model <model> is used in a coupled connector list but has no corresponding symmetric connector. A coupled pragma is missing or does not have all the coupled connectors declared.

while swapping <conname1> and <conname2>, one of the connector did not have coupled connector list while the other has

In the model, a coupled pragma is missing for either connector <conname1> or connector <conname2>.

same signal in different symmetry list

The same connector was encountered in two different symmetric connector lists. All the symmetric connectors must be put in the same symmetric connector list.

same signal in different coupled list

The same connector was encountered in two different coupled connector list. All the coupled connectors must be put in the same coupled connector list.

Only one variable authorized in a 'for' expression. Use Hierarchy!

This means there is a loop in the model that contains more than one generic variable of unknown value. This is forbidden. The user must use a separate hierarchical level for each of the variables.

Index