cadence®

Virtuoso Unified Custom Constraints Configuration Guide

Product Version IC23.1
September 2023

Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Contents

1

Customizing Constraint Types Using a Configuration File

The Cadence® Virtuoso® unified custom constraint management system allows you to establish design needs, save them as constraints, and share those constraints across specification, simulation, and implementation to drive the accelerated layout solution with reduced errors. A constraint-driven design preserves the design intent by enabling efficient design collaboration.

The unified custom constraint management system is available in:

- Virtuoso® Schematic Editor XL (Schematics XL)
- Virtuoso® Layout Suite XL (Layout XL)
- Virtuoso® Layout Suite GXL (Layout GXL) **(IC6.1.8 Only)**

For more information, see the *Virtuoso Unified Custom Constraints User Guide*.

This configuration guide describes how to customize, configure and extend Cadence® Virtuoso® unified custom constraints (constraints) functionality in the Cadence® IC releases.

This configuration guide is aimed primarily at on-site CAD departments and assumes that you are familiar with:

- The Virtuoso Studio Design Environment and application infrastructure mechanisms designed to support consistent operations between all Cadence® tools.
- The applications used to design and develop integrated circuits in the Virtuoso Studio Design Environment, notably, the Virtuoso® Layout Suite, and Virtuoso® Schematic Editor.
- The Virtuoso Studio Design Environment technology file.
- Component description format (CDF), which lets you create and describe your own components for use with Layout XL.

In addition to creating constraints based on default system constraint types (Cadence-provided built-in constraint types), you can create new custom constraint types, override definitions of existing custom constraint types, and extend definitions of any system constraint type using the UI configuration file, `config.xml`. This file provides complete information about default and custom constraints, including their GUI displays.

The following topics in this chapter discuss the components of a `config.xml` file and how to use them to customize the system constraint types or create new custom constraint types:

- An Introduction to a Configuration File
- Understanding the XML Tags Used in a Configuration File
- Extending System Constraint Types
- Creating and Overriding Custom Constraint Types
- Creating New Versions of a Constraint Type
- Using Multiple Constraint Configuration Files
- Using Validation and Verification Callbacks
- Best Practices for Custom Constraint Type Definitions
- Adding a New Icon to the Constraint Manager Assistant
- Propagating Constraints to the Checks/Asserts Assistant
- Examples of Using the config.xml File

Licensing Requirements

Licensing Requirements for Virtuoso Unified Custom Constraints

To use the constraint functionality in:

Virtuoso Tool	License Required
Schematic XL	`Virtuoso_Schematic_Editor_XL` (License Number 95115)
Layout XL	`Virtuoso_Layout_Suite_XL` (License Number 95310)
Layout GXL	`Virtuoso_Layout_Suite_GXL` (License Number 95321) **(IC6.1.8 Only)**

For information on licensing in the Virtuoso Studio Design Environment, refer to *Virtuoso Software Licensing and Configuration User Guide*.

Depending on the advanced nodes feature being used, you will need one of the following licenses, in addition to the base product license:

- `Virtuoso_Adv_Node_Opt_Lay_Std` (License Number 95512)
- `Virtuoso_Adv_Node_Opt_Layout` (License Number 95511)

For more information, see License Requirements for Advanced Node Features in *Virtuoso Software Licensing and Configuration User Guide*.

Licensing Requirements for PVS-CV

The following license is required for running PVS-CV in Virtuoso:

- `Phys_Ver_Sys_Const_Validator` (license number 96300)

Note: The PVE12.1.1 or later version should be used for the PVS-CV feature.

An Introduction to a Configuration File

The UI configuration file, `config.xml`, lets you manage the definition of constraint types including members, parameters, and other callbacks as well as the display of the constraints and parameters in the *Constraint Manager* assistant. By default, the definition of Cadence-provided system constraint types are stored in the `config.xml` file that can be found at the following path:

```
$CDSHOME/share/cdssetup/dfII/ci/config.xml
```

However, you can override this file by creating a new `config.xml` file that contains the desired customizations. If you save this new file in the `.cadence/dfII/ci` directory located on the Cadence Setup Search File (CSF) path, it will get picked automatically at the time of *Constraint Browser* initialization. If you have saved the `config.xml` file at a different location, use the ciLoadConfigXML SKILL function to load it at run time. However, the caveat here is that a `config.xml` file which is not on the CSF path needs to be loaded every time you start a new session of Virtuoso.

When the `config.xml` file containing the definition of a new constraint type is loaded, the *Constraints Manager* assistant displays the new constraint type in the *Constraint Generator* drop-down menu. When you create a constraint using this new constraint type, all associated parameters are displayed in the *Constraint Parameter Editor*.

In addition to creating new constraint types and its associated parameters, you can use a `config.xml` file to control their display in the *Constraint Manager* assistant. For example, you can rename a constraint type, hide a constraint type from the *Constraint Generator* drop-down menu, or control the display format of a constraint parameter.

Understanding the XML Tags Used in a Configuration File

For creating, extending, overriding, and formatting constraint types in a `config.xml` file, use the following XML tags:

- ConstraintType
- Name (Constraint Type Name)
- GUIName (Constraint Type Display Name)
- Version
- BaseType
- Categories
- CustomSimulationCheckCB

- Param
  - Name (Constraint Parameter Name)
  - GUIName (Constraint Parameter Display Name)
  - Type
  - DefaultValue
  - ValueRange
  - Scope
- MinMembers
- MaxMembers
- AllowedMemberTypes
- VerifyCB
- ValidateCB

A typical constraint type definition in the `config.xml` file looks like following:

```
<ConstraintType>
    <Name>myCustomConstraint</Name>
    <GUIName menu="Placement(DEFAULT)">myCustomConstraint</GUIName>
    <Param>
        <Name>myCustomParam</Name>
        <GUIName summary="ValOnly">myCustomParam</GUIName>
        <Type>int</Type>
        <Scope>constraint</Scope>
        <DefaultValue>4</DefaultValue>
        <ValueRange>0 10</ValueRange>
    </Param>
    <MinMembers>1</MinMembers>
    <MaxMembers>2</MaxMembers>
    <AllowedMemberTypes>inst,net,instTerm,pin,cluster,modgen,shape,guide</
    AllowedMemberTypes>
    <VerifyCB>CUSTExampleVerifyCB</VerifyCB>
    <ValidateCB>CUSTExampleValidateCB</ValidateCB>
</ConstraintType>
```

ConstraintType

(Mandatory) Defines the name and parameters of a constraint type. The body of this XML tag contains the definitions of other XML tags explained in the sections below.

Name (Constraint Type Name)

(Mandatory) Defines the name of the constraint type, as shown below.

`<Name>`*`ConstTypeName`*`</Name>`

Note: It is recommended that you use a private prefix starting with an upper case, such as `Cst` for Customer.

GUIName (Constraint Type Display Name)

Defines the display name of the constraint type.

Note: If you do not specify the `<GUIName>` tag, the constraint type name specified with the `<Name>` tag is used as the GUI display name.

The `<GUIName>` tag can have the following attribute:

`menu` Controls the placement of the constraint type in the *Constraint Generator* drop-down menu. It accepts the following values:

`submenuName`

The style of specifying this value determines how and where the constraint type will be displayed in the *Constraint Generator* drop-down menu. For information about the different styles and their impact on the GUI, see Styles of Using submenuName with the menu Attribute.

`subMenuName``(DEFAULT)`

When you specify `(DEFAULT)` after the *`submenuName`*, the constraint type gets added to the specified submenu and becomes available for selection by default when you open the *Constraint Manager* assistant. For example:

```
<GUIName menu="Electrical(DEFAULT)">myCustomConstraint</GUIName>
```

`{HIDE}`

When you specify `{HIDE}`, the associated constraint type gets hidden from the *Constraint Generator* drop-down menu. For more information, see Hiding Constraint Types in the Constraint Generator Menu.

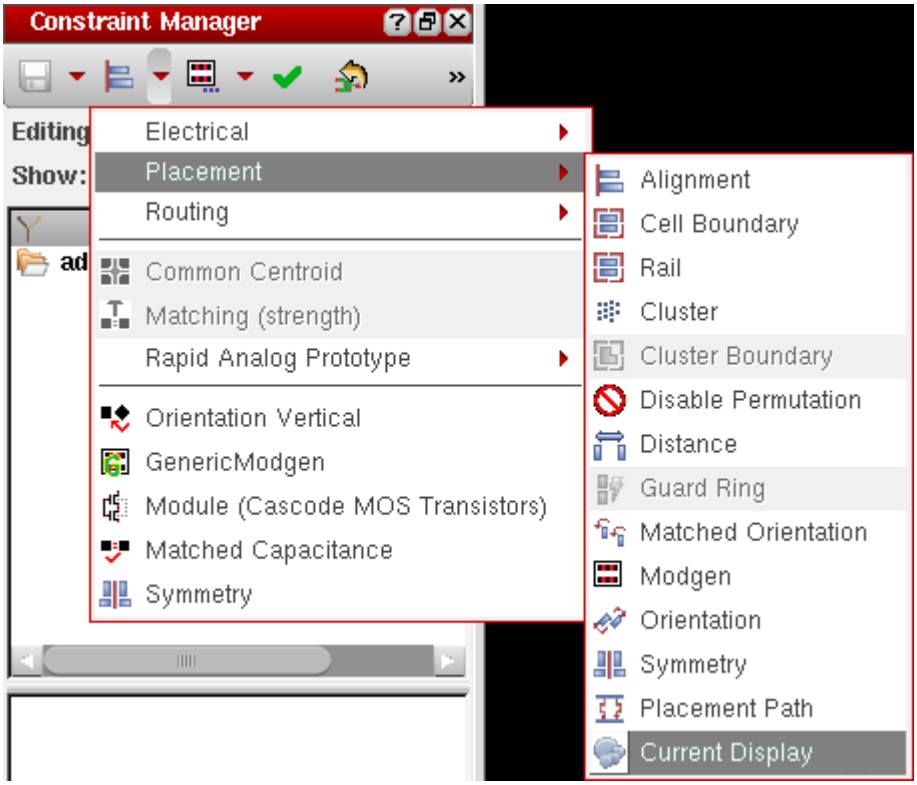
Styles of Using submenuName with the menu Attribute

With the `menu` attribute, you can specify a submenu name in different styles to determine how it should be displayed, that is, whether as new submenu, under an existing submenu, or as a hierarchical menu.

- Specify the name of a submenu that already exists in the *Constraint Generator* drop-down menu. The new constraint type will get added as an option below that submenu. For example, if the new constraint type, *Current Display*, should be displayed in the existing Cadence-provided submenu, *Placement*, specify the following:

  ```
  <GUIName menu="Placement">Current Display</GUIName>
  ```

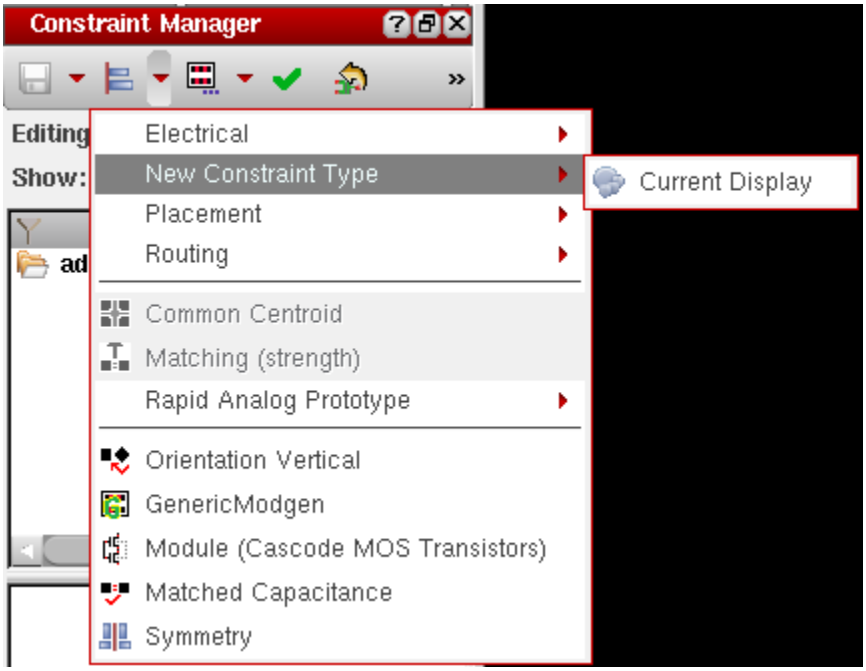
  The new constraint type will be displayed as follows:



- If the specified submenu does not exist, a new submenu by the given name gets created the *Constraint Generator* drop-down menu and the constraint type gets added below it. For example, if you want a new submenu, *New Constraint Type*, where the new constraint type, *Current Display*, should be displayed, specify the following:

  ```
  <GUIName menu="New Constraint Type">Current Display</GUIName>
  ```

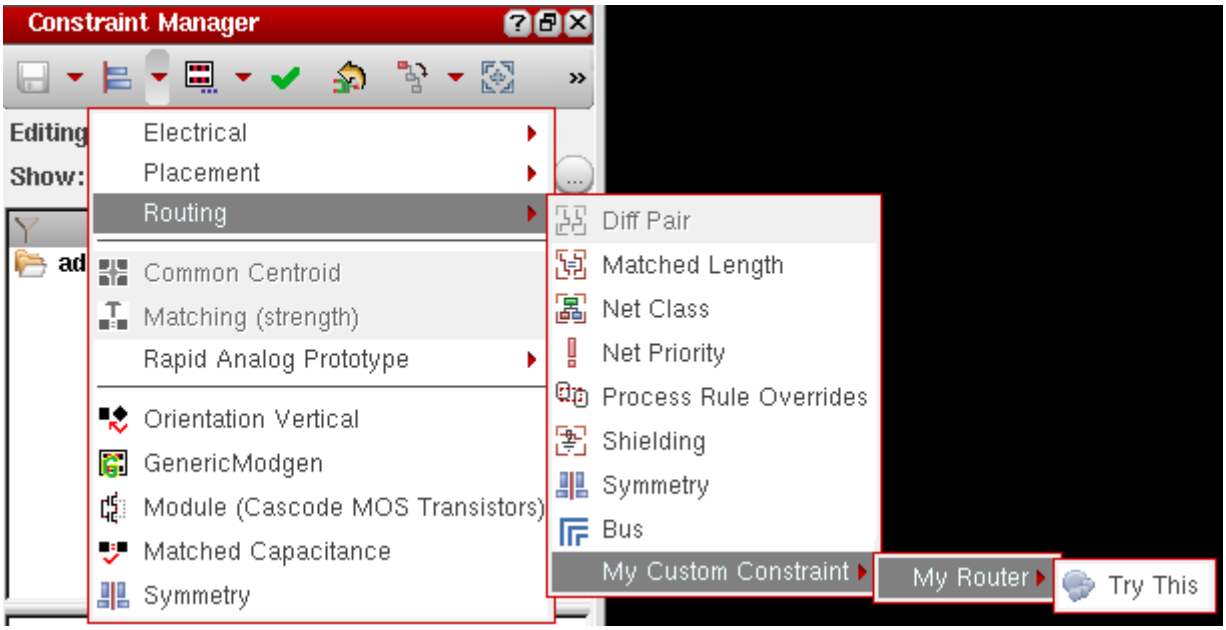
The new constraint type will be displayed as follows:



- To create a hierarchy of menus, use forward slashes (`/`) as separators while specifying a list of submenu names. For example, if you want to add the new constraint type, *Try This*, to be displayed below *Routing* (which is an existing Cadence-provided submenu) – *My Custom Constraint* – *My Router*, specify the following:

```
<GUIName menu="Routing/My Custom Constraint/My Router">Try This</GUIName>
```

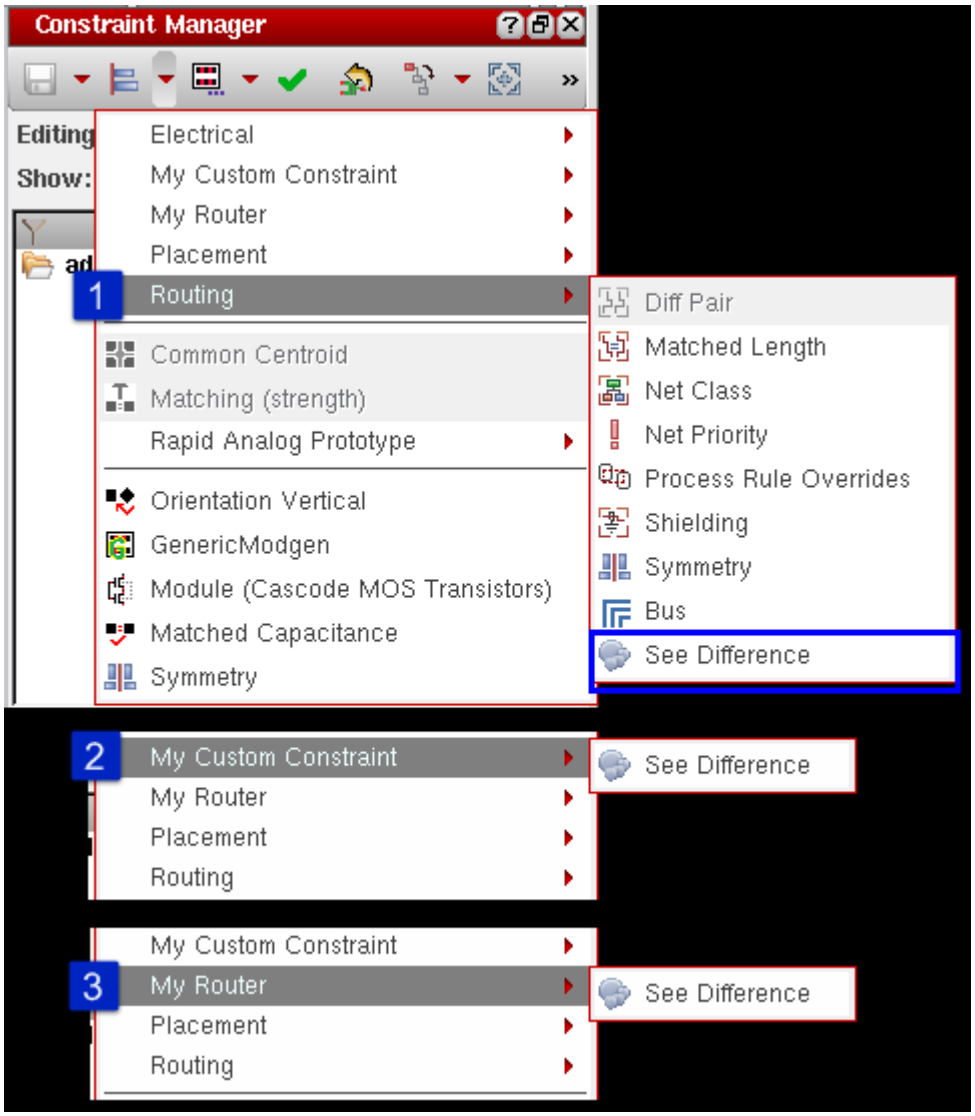
The new constraint type will be displayed as follows:



- To add a constraint type to multiple submenus, use colon (:) as separators while specifying a list of submenu names. For example, you want to add the new constraint type, *See Difference*, to be displayed below three submenus, which includes an existing Cadence-provided submenu, *Routing*, and two new user-defined submenus, *My Custom Constraint* and *My Router*. In this case, specify the following:

```
<GUIName menu="Routing:My Custom Constraint:My Router">See Difference</
GUIName>
```

The new constraint type will be displayed as follows:



Hiding Constraint Types in the Constraint Generator Menu

If needed, you can hide constraint types such as *Symmetry*, *Orientation*, and *Alignment* from the *Constraint Generator* menu in the Constraint Manager assistant. To do so, in the `config.xml` file, specify `"{HIDE}"` as the `menu` name in the `GUIName` tag for the constraint that needs to be hidden. The example below shows how to hide the *Alignment* constraint from the menu.

```
<ConstraintConfig>
<ConstraintType>
    <Name>alignment</Name>
```

```
    <GUIName menu="{HIDE}">Alignment</GUIName>
    ...
    <Param>
        <Name>Owner</Name>
        <GUIName visibleInEditor="false">Owner</GUIName>
    </Param>
   ...
    <Param>
        <Name>edgeNumber</Name>
        <GUIName visibleInEditor="false">Edge</GUIName>
    </Param>
    ...
</ConstraintType>
...
</ConstraintConfig>
```

Note: A separate `config.xml` file can also be created to hide specific constraint types from the menu and can be loaded using the `ciLoadConfigXML` function. Alternatively, you can use the `ciLoadConfigXMLFromString` function that does not require loading the `config.xml` files from certain search paths on the disk. With this capability of loading the `config.xml` files from a string, the `config.xml` strings can reside in SKILL files and be encrypted, if necessary.

Version

(Optional) Specifies the version number of the constraint when multiple versions of a constraint exist. This tag is used in conjunction with the BaseType tag allowing you to define multiple versions of a custom constraint type.

Note: Each version must have its own unique Constraint Type Name and share the same BaseType name to allow multiple versions of the same base constraint type to exist at the same time. For more details, see Creating New Versions of a Constraint Type.

BaseType

(Optional) Specifies the original constraint name. This tag is used in conjunction with the Version tag allowing you to define multiple versions of a custom constraint type based on each sharing the same BaseType.

Note: Each version must have its own unique Constraint Type Name and share the same BaseType name to allow multiple versions of the same base constraint type to exist at the same time. For more details, see Creating New Versions of a Constraint Type.

Categories

(Optional) Specifies the category of constraint, as shown below.

```
<Categories>frontEnd,Physical</Categories>
```

Any custom or default system constraint created in the Constraint Manager assistant can be propagated automatically to the Virtuoso ADE Assembler the *Checks/Asserts assistant*, by including the value, `customAssertion`, in the <Categories> tag within the `config.xml` file, as shown in the example below.

```
<Categories>frontEnd,Physical,customAssertion</Categories>
```

For more details refer to Propagating Constraints to the Checks/Asserts Assistant.

CustomSimulationCheckCB

(Optional) For constraints belonging to the category `customAssertion`, specifies a custom callback in the `config.xml` file, as shown below.

```
<CustomSimulationCheckCB>maxCurrentCustomCheckCallback</
CustomSimulationCheckCB>
```

This allows `customAssertion` constraints to be netlisted as Spectre asserts along with the existing checks and asserts. For more details refer to Propagating Constraints to the Checks/Asserts Assistant.

Param

Defines the constraint parameters. Within the body of the `ConstraintType` tag, you can define multiple `Param` tags based on your requirement. The parameters defined in each body of a `Param` tag are added to a list of parameters that can be used for the given constraint type.

The following tags are defined within the body of a `Param` tag:

- Name (Constraint Parameter Name)
- GUIName (Constraint Parameter Display Name)
- Type
- DefaultValue
- ValueRange

- Scope

Name (Constraint Parameter Name)

Specifies the constraint parameter name. Ensure that a constraint parameter name is always unique and not reused.

Note: This tag is mandatory if the `Param` tag has been specified.

GUIName (Constraint Parameter Display Name)

Defines the GUI display name of the constraint parameter.

Note: If you do not specify the `<GUIName>` tag, the constraint parameter name specified with the `<Name>` tag is used as the GUI display name.

The `<GUIName>` tag can have the following attributes:

`visibleInEditor` Controls the display of the constraint parameters in the *Constraint Parameter Editor* pane of the *Constraint Manager* assistant. It accepts the following values:

- `true` Displays a constraint parameter in the *Constraint Parameter Editor* pane. This is the default value.
- `false` Hides a constraint parameter in the *Constraint Parameter Editor* pane.

`summary` Controls how the constraint parameters should be summarized in the *Parameters* column in the *Constraint Browser*. This attribute accepts the following values:

- `Ignore` Does not display the constraint parameter summary in the *Parameters* column. This is the default value of the `summary` attribute.
- `NameAndVal` Displays the name and value of the constraint parameter as "`name=value`".
- `ValOnly` Displays only the value of the constraint parameter.
- `ValOnlyIfNotDefault` Displays only the value of the constraint parameter if it is not the default value.

`NameAndValIfNotDefault`

Displays the name and value of the constraint parameter as "`name=value`" only if the value is not the default.

`NameOnlyIfTrue`

Displays the name of the constraint parameter if its value is "`True`".

Note: This attribute value is valid only for Boolean parameters.

Type

Defines the type of values the constraint type can accept. This tag accepts one of the following keywords: `boolean`, `int`, `double`, `string`, `enum`, or `enumset`.

Note: This tag is mandatory if the `Param` tag has been specified.

ValueRange

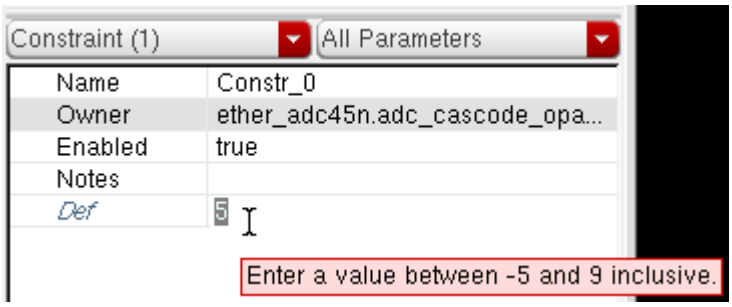
Defines the value range based on the parameter type, as given below.

Note: If `<Type>` is `boolean` or `string`, you do not need to specify `<ValueRange>`.

- When `<Type>` is `int` or `double`, the `<ValueRange>` tag specifies the minimum and maximum acceptable value for the parameter. For example, the following definition in the `config.xml` file:

  ```
  <ValueRange>-5 9</ValueRange>
  ```

  will be translated in the *Constraint Parameter Editor* pane as shown in below.



  You can also specify multiple acceptable values in any order. The minimum and maximum values out of the given values will be picked automatically to define the range.

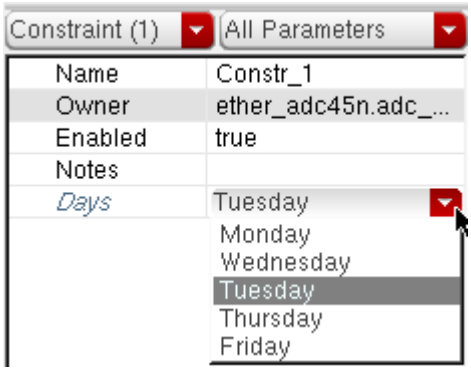
For example, from the following `ValueRange` statement, `9` will be considered as the maximum value and `-5` as the minimum value:

```
<Type>int</Type>
<ValueRange>0 7 9 -5</ValueRange>
```

- When `<Type>` is `enum` , the `<ValueRange>` tag specifies the acceptable values for the parameter. The specified values appear in a drop-down list box and only one value out of those can be the default. For example, the following definitions in the `config.xml` file:

  ```
  <Type>enum</Type>
  <ValueRange>Monday Wednesday Tuesday Thursday Friday</ValueRange>
  <DefaultValue>Tuesday</DefaultValue>
  ```

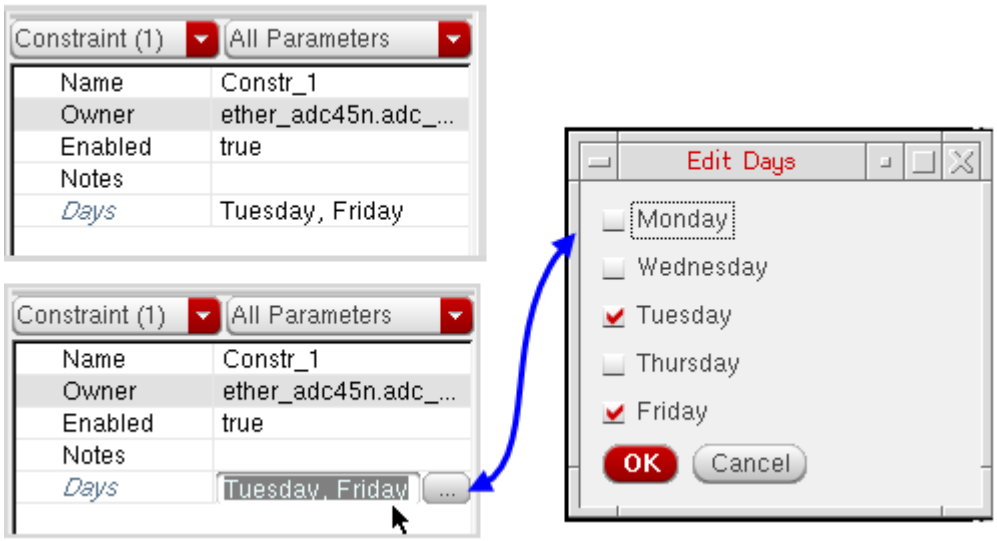
  will be translated in the *Constraint Parameter Editor* pane as shown in below.



- When `<Type>` is `enumset`, the `<ValueRange>` tag specifies the acceptable values for the parameter. In this case, a field auto-populated with the default value is displayed adjacent to the parameter name. When you click this field, a button is displayed on its right. Clicking the button opens a form containing all acceptable values defined for the parameter with check boxes corresponding to each value to allow selection of the required values. For example, the following definitions in the `config.xml` file:

  ```
  <Type>enumset</Type>
  <ValueRange>Monday Wednesday Tuesday Thursday Friday</ValueRange>
  <DefaultValue>Tuesday Friday</DefaultValue>
  ```

will be translated in the *Constraint Parameter Editor* pane as shown in below.



DefaultValue

Defines the default value of a parameter.

Note: For parameters that have a `DefaultValue` tag, but no `ValueRange` tag, the specified `DefaultValue` and `ValueRange` are considered to be the same. However, for parameters of `int` or `double` type, the allowed `ValueRange` is between `-999999999` to `999999999`.

In addition, the following rules are applicable to definition of the `DefaultValue` tag:

- For `int` and `double` type, the minimum of `ValueRange` will be considered as the default value in the following scenarios:
  - The specified default value does not fall in the range of the given `ValueRange`.
  - No default value was specified.
- For `boolean` type, if a `DefaultValue` is not specified, `false` is selected by default in the parameter's drop-down list box.
- For `string` type, there are no restrictions related to default value. If nothing is specified with the `<DefaultValue>` tag, the default is taken to be blank.

Scope

Restricts the scope of a parameter, that is, defines whether a parameter is a constraint parameter or member parameter. It has the following syntax:

`<Scope>`*`validScopeValue`*`</Scope>`

This tag accepts one of the following valid values:

`constraint`	Consider the associated parameter as constraint parameter only.
`member`	Consider the associated parameter as member parameter only.
`constraint,member`	Consider the associated parameter as both constraint parameter and member parameter.

If no value is given or the tag is missing, the constraint is treated as both constraint parameter and member parameter.

MinMembers

Specifies the minimum number of members needed for a constraint type.

The `MinMembers` tag defaults to `1`, when one of the following conditions is true:

- The values specified within the `AllowedMemberTypes` tag include `master`.
- The values specified within the `AllowedMemberTypes` tag do not include `master`.

The `MinMembers` tag defaults to `0` when only `master` is specified within the `AllowedMemberTypes` tag.

MaxMembers

Specifies the maximum number of members allowed for a constraint type. If the `MaxMembers` tag is not specified, by default, any number of members greater than or equal to `MinMembers` are allowed.

AllowedMemberTypes

(Mandatory) Specifies the type of members (objects) that can be included as part of your custom constraint. For defining the `allowedMemberTypes` tag, you can use one or more of

the following values in any combination: `inst`, `master`, `net`, `instTerm`, `pin`, `netTerm`, `netClass`, `modgen`, `boundary`, `cluster`, `shape`, `via`, and `guide`.

You must complete this tag otherwise an error message will be displayed.

VerifyCB

Defines the name of the verify callback. It is a one word string. For example:

```
<VerifyCB>callbackName</VerifyCB>
```

See also Using Validation and Verification Callbacks.

ValidateCB

Defines the name of the validate callback. It is a one word string. For example:

```
<ValidateCB>validateCBName</ValidateCB>
```

See also Using Validation and Verification Callbacks.

IgnoreConstraintType

Controls whether a constraint type should be ignored when the `config.xml` file is read. It is a Boolean value, which is set to `false` by default. If this tag is set to `true` in the body of a custom constraint type definition, the constraint type is not created.

You can also use this tag within the constraint generator definitions, which might be present in the `config.xml` file for GUI customizations. This tag prevents creation of a custom constraint type for those constraint generators.

Extending System Constraint Types

To extend system constraint types, you can perform only the following actions:

- Update the `GUIName` tag and the `menu` attributes (see GUIName (Constraint Type Display Name))
- Add or update parameters (see Param)
- Update VerifyCB and ValidateCB

The extended system constraint types can be loaded at startup, that is, before a Virtuoso session is launched, or during a running session.

If you have extended an existing system constraint type, the related modifications get merged to the corresponding entries in the `config.xml` file. If a constraint type is read for the first time from a `config.xml`, it gets added to the in-memory configuration. When an entry of the same constraint type is detected in the same `config.xml` file or any other `config.xml` file on the CSF path, its unique tags and parameters are appended to the constraint type already present in the in-memory configuration. For more information on CSF path, see The Cadence Setup Search File: setup.loc in the *Cadence Application Infrastructure User Guide*.

See also Example 1: Extending an Existing System Constraint Type.

At Startup

To extend a system constraint type at startup:

1. Create a `config.xml` file in a `.cadence/dfII/ci` directory located on the CSF path.
2. Incorporate the required updates for the system constraint type you want to extend.

When Virtuoso is launched, all `config.xml` in CSF paths are read and the relevant (unique) tags for the existing constraint type are appended to the in-memory configuration. If new parameters were added to `config.xml`, then at the startup, the parameters with unique names are picked and appended to the existing configuration.

During a Virtuoso Session

To extend an existing system constraint type during a running Virtuoso session:

1. Create a `config.xml` file.
2. Incorporate the required updates for the system constraint type you want to extend.
3. Use the `ciLoadConfigXML` SKILL function to specify the path of the new `config.xml` file as an argument. This modifies the impacted system constraint type(s) instantly.

Tip

Alternatively, you can use the `ciLoadConfigXMLFromString` function that does not require loading the `config.xml` files from certain search paths on the disk.

Creating and Overriding Custom Constraint Types

Creating New Custom Constraint Types

Custom constraint types can be created at startup, that is, before a Virtuoso session is launched, or during a running session.

See also Example 2: Creating a New Custom Constraint Type.

At Startup

To create a new constraint type at startup:

1. Create a `config.xml` file in a `.cadence/dfII/ci` directory located on the CSF path.
2. Add to this file a unique constraint type entry.

When Virtuoso is launched, all `config.xml` in CSF paths are read and therefore, the custom constraint type is created.

During a Virtuoso Session

To create a new constraint type during a running Virtuoso session:

1. Create a `config.xml` file.
2. Add to this file the required constraint type entry.
3. Use the `ciLoadConfigXML` SKILL function to specify the path of the new `config.xml` file as an argument.

Tip

Alternatively, you can use the `ciLoadConfigXMLFromString` function that does not require loading the `config.xml` files from certain search paths on the disk.

The newly created custom constraint type is available for use instantly.

Overriding Existing Custom Constraint Types

The existing custom constraint types can be overridden with new definition at startup or during a running Virtuoso session. In this case, the definition is not extended as is the case

with Cadence-provided system constraint types. Instead, their definition is completely overridden with the new definition.

Note: For custom constraint types still created using `propdict.def`, overriding does not work. They will be treated as system constraint type and cannot be extended or overridden using a new `config.xml` file definition.

See also Example 3: Overriding an Existing Custom Constraint Type.

At Startup

To override a custom constraint type at startup:

1. Create a `config.xml` file in a `.cadence/dfII/ci` directory located on the CSF path.
2. Add to this file the new or updated parameters for an existing custom constraint type.

When Virtuoso is launched, all `config.xml` files in the CSF paths are read and the latest tags for a specific custom constraint type override its existing definitions, if already present in the in-memory configuration. Therefore, the last found definition of a custom constraint type is honored.

During Virtuoso Session

To override an existing custom constraint type during a running Virtuoso session:

1. Create a `config.xml` file.
2. Add to this file the new or updated parameters for an existing custom constraint type.
3. Use the `ciLoadConfigXML` SKILL function to specify the path of the new `config.xml` file as an argument. This overrides the impacted custom constraint type(s) instantly.

Tip

Alternatively, you can use the `ciLoadConfigXMLFromString` function that does not require loading the `config.xml` files from certain search paths on the disk.

Creating New Versions of a Constraint Type

You can create a new version of a constraint type, allowing multiple versions of a constraint type to exist while preserving the constraints that currently use an existing version.

The tags *BaseType* and *Version* let you control the different versions of your customized constraint types. Before creating different versions of a constraint type, you must first protect the existing constraint type by renaming it with a unique *Name* and giving it a *BaseType* and *Version* tag. You can then create a new version of the constraint type using the original constraint type name and giving it a unique version number while sharing the same *BaseType*.

Important

If *BaseType* and *Version* tags are not created for an existing constraint before creating a new version, the new version created replaces all instances where the previous constraint type was used. Any differences between the new and old version will not be preserved.

Creating a New Version of a Constraint Type

You can replace an existing version of a custom constraint type with a new version, making the existing version obsolete while preserving the definitions used by existing constraints.

1. Identify the existing custom constraint type required. For example, `CurrentDisplay`.
2. Give the constraint type a new unique *Name*. For example, `CurrentDisplay_v1`.
3. In *BaseType* enter the original name of the constraint type. For example, `CurrentDisplay`.
4. In *Version* enter `1`.
5. (optional) Rename the constraint type's *GUI Name* to distinguish this version of the constraint type from other versions displayed in the system. For example, `CurrentDisplay_v1`.

   The original version of this constraint type is now protected.
6. Create a new constraint type and give it the original constraint type name. For example, `CurrentDisplay`.

   Note: Creating a new constraint type with the same name will override the current constraint type.

7. In *BaseType* enter the same BaseType as that selected for the original version of the constraint type. For example, `CurrentDisplay`.
8. Set the *Version* to `2`.
9. (optional) Rename the constraint type's *GUI Name* to distinguish this version of the constraint type from other versions displayed in the system. For example, `CurrentDisplay_v2`.

   The examples above would result in two versions of a constraint type with a base type of `CurrentDisplay`:

Constraint Type Name	GUI Name	Version	Base Type
CurrentDisplay_v1	CurrentDisplay_v1	1	CurrentDisplay
CurrentDisplay	CurrentDisplay_v2	2	CurrentDisplay

10. Create any additional versions of the constraint type if required, incrementing the *Version* each time.

You can use the SKILL functions ciTypeDefBaseType and ciTypeDefVersion to return the version number and base type of a constraint type. You can also use these functions with the VerifyCB callback to determine if an existing constraint is an older version and can also be used to disable older constraint types. For more details on the VerifyCB callback, see Using Validation and Verification Callbacks.

Using Multiple Constraint Configuration Files

You can create multiple constraint configuration (`config.xml`) files and choose to save them on the CSF search path or elsewhere. If multiple `config.xml` files are found on the CSF search path, they will be loaded by the constraints system one-by-one, in append mode, based on the following rules:

- If any configuration file fails to load due to, for example, a parsing error, a warning will be displayed before proceeding to load the remaining configuration files.
- If a distant configuration file has defined a particular `ConstraintType`, while a closer configuration has not defined it, the `ConstraintType` will be maintained in the current configuration (as the configuration files will be merged).

  Note: Here, *closer* refers to a path that can be found earlier in the path list, while *distant* is a path located at any position after the *closer* path. Therefore, *closer* and *distant* refer to how early, or late, that particular path appears in the "path list" used by the search mechanism.

- If two configuration files have defined the same system constraint type, but with different parameters, the mutually-exclusive parameters will be merged. However, for custom constraint types, if a corresponding definition already exists and you load another definition, then the newer definition overwrites the previous one. This means that the custom constraint type definitions can never be merged.

Using Validation and Verification Callbacks

In the `config.xml` file, you can use of the following two callbacks related for validating and verifying new and modified constraint types:

```
<VerifyCB>verifyCBName</VerifyCB>
<ValidateCB>validateCBName</ValidateCB>
```

Validating Constraints

The validate callback is called automatically on the layout side whenever a constraint is created or modified (as with constraint checking) and is used by CAE (Constraint Aware Editing) during interactive edits to test and mark the constraint as being *enforced*.

Note: The validate callback is not called on the schematic side.

You can insert a constraint validation callback into the body of ConstraintType XML tag in the `config.xml` file. For example:

```
<ConstraintType>
    <Name>symmetry</Name>
    <ValidateCB>CstSymmetryCallback</ValidateCB>
</ConstraintType>
```

- The SKILL function (`CstSymmetryCallback` in this case) must be defined at some point during the current working session.
- The `validateCB` functions must be of the format *`function_name`*`(`*`d_cellViewg_constraintID`*`)` and:
  - have two arguments. The first being the layout cellview ID and the second being the constraint ID to be validated.
  - must return `t` or `nil`, corresponding to whether the constraint is enforced or not enforced.

When the above validation callback is set in a `config.xml` file, any symmetry constraint on the layout will use the user-defined "`CstSymmetryCallback`" function when the Constraint Aware Editing (CAE) validation code is run. Constraint status will also be updated when the

Verify – Design command is run in Layout XL (and higher tiers) (the *Process Rules* tab is used to configure verification settings).

Verifying Constraints Consistency

To verify the consistency of the constraint members and parameters, the verify callback is called automatically when a constraint of a related type is created or modified. When a new constraint is created it will have all the attributes, members, and parameters properly populated. However, it will not be listed for those objects that refer to it until a constraint verification callback succeeds. The verify callback should use a SKILL API or the UT fields to *verify* whether the members and parameters are consistent. The SKILL callback has one single argument, the constraint ID.

If the callback returns `nil`, the verification fails with a warning message in the CIW, and the modification or creation is rejected. Any other value returned, including "`FALSE`", is considered as a non-nil return and the changes will be accepted. See also Constraints Checks in the *Virtuoso Schematic Editor User Guide*.

A typical usage of verification callback is to yield warning messages and return `nil` when invalid combination of parameter settings and member is used. For example, your verify callback can allow a parameter to be set on a certain type of member, say net member, or on the first member. Moreover, it can restrict that only the first member can be, for example, an instance. Any logical check, with respect to constraint members and parameters, can be performed using the `VerifyCB` callback to restrict parameter and member usage.

Example for VerifyCB

The following is an example of the use of `VerifyCB`, as set in the `config.xml` file.

In a config.xml file:

```
<ConstraintType>
    <Name>ExampleCon</Name>
    <VerifyCB>CUSTExampleConVerifyCB</VerifyCB>
    <Param>
        <Name>param1</Name>
        <ValueType>double</ValueType>
        <DefaultValue>0.5</DefaultValue>
        <ValueRange>0.0 1.0</ValueRange>
    </Param>
</ConstraintType>
```

In a SKILL file loaded either in .cdsinit or in library libInit.il:

```
procedure(CUSTExampleConVerifyCB(con)
  ; This procedure validates that the right members and parameters are set on the
constraint.
  let(((retVal t))
    when(length(ciConListMembers(con)) < 2
     warn("Constraint type %s requires at least 2 members. Constraint %s has only
%d members.\n" ciConGetType(con) ciConGetName(con) length(ciConListMembers(con)))
      retVal = nil
    )
    foreach(mem ciConListMembers(con)
      when(cadr(mem) != 'inst
        warn("Constraint type %s requires instances as members. Member %s is of
type %s.\n" ciConGetType(con) car(mem) cadr(mem))
        retVal = nil
      )
    )
    retVal
  )
)
```

Best Practices for Custom Constraint Type Definitions

When creating a custom constraint type, the following should be considered:

- The database requires constraint parameter definitions to be unique and reserves some names. A conflict might occur if the same constraints, templates, or reserved parameters are specified with different types and a message will be displayed in the CIW.
- All constraint and template parameter definitions are checked for mismatched types at initialization and when the Constraint Manager assistant is opened. Prefixing parameters names with a unique identifier can be useful to avoid conflicts. For example, you could use your company name and constraint name as a prefix to make the constraint parameter names unique.

Adding a New Icon to the Constraint Manager Assistant

When you create a new constraint type, you also need an associated icon that will be displayed adjacent to it in the *Constraint Manager* assistant. This new icon should be an image of 16x16 bit size that is saved with a name of the format: `<constraintName>.png`

When you move the mouse cursor over the new icon in the *Constraint Manager*, a tooltip will display the text as defined in the `config.xml` file.

For your reference, a few examples are stored in the Cadence installation directory under:

`./share/cdssetup/icons/16x16`

For the *Constraint Manager* assistant to load a `.png` file, it must be located in the following directory:

`.cadence/icons/16x16`

Note: The `.cadence` directory must be on the file search path. For more information, see The Cadence Setup Search File: setup.loc *Cadence Application Infrastructure User Guide*.

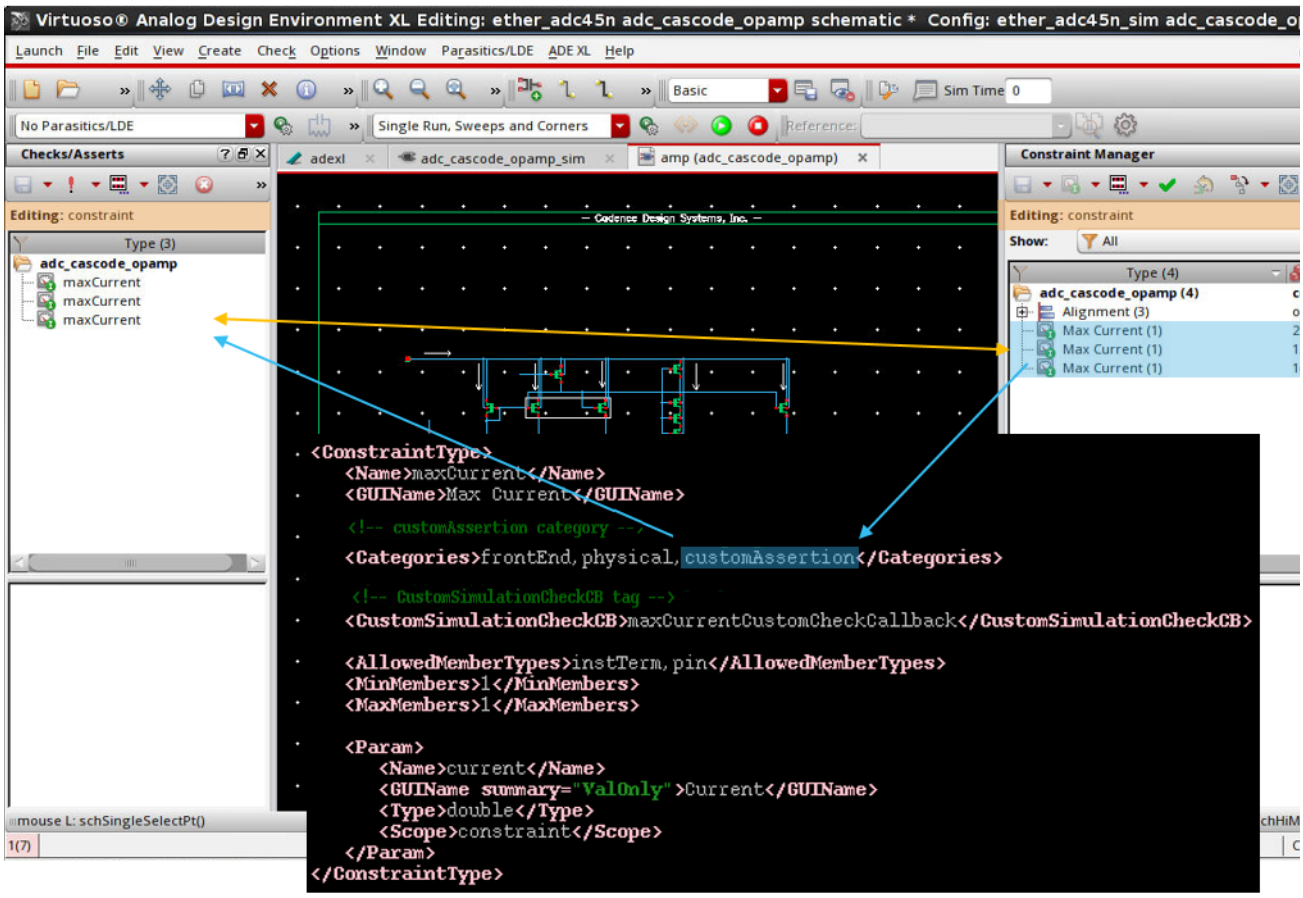
Propagating Constraints to the Checks/Asserts Assistant

Any custom or default system constraint created in the Constraint Manager assistant can be propagated automatically to the Checks/Asserts assistant, which is accessible from the Virtuoso ADE product suite.

To enable this automatic propagation, update the `<ConstraintType>` definition in the `config.xml` file to include the value, `customAssertion`, in the `<Categories>` tag, as shown in the example below.

```
<ConstraintType>
   <Name>maxCurrent</Name>
   <GUIName>Max Current</GUIName>

   <!-- customAssertion category -->
   <Categories>frontEnd,physical,customAssertion</Categories>

   <!-- CustomSimulationCheckCB tag -->
   <CustomSimulationCheckCB>maxCurrentCustomCheckCallback</CustomSimulationCheckCB>

   <AllowedMemberTypes>instTerm,pin</AllowedMemberTypes>
   <MinMembers>1</MinMembers>
   <MaxMembers>1</MaxMembers>

   <Param>
      <Name>current</Name>
      <GUIName summary="ValOnly">Current</GUIName>
      <Type>double</Type>
      <Scope>constraint</Scope>
   </Param>
</ConstraintType>
```

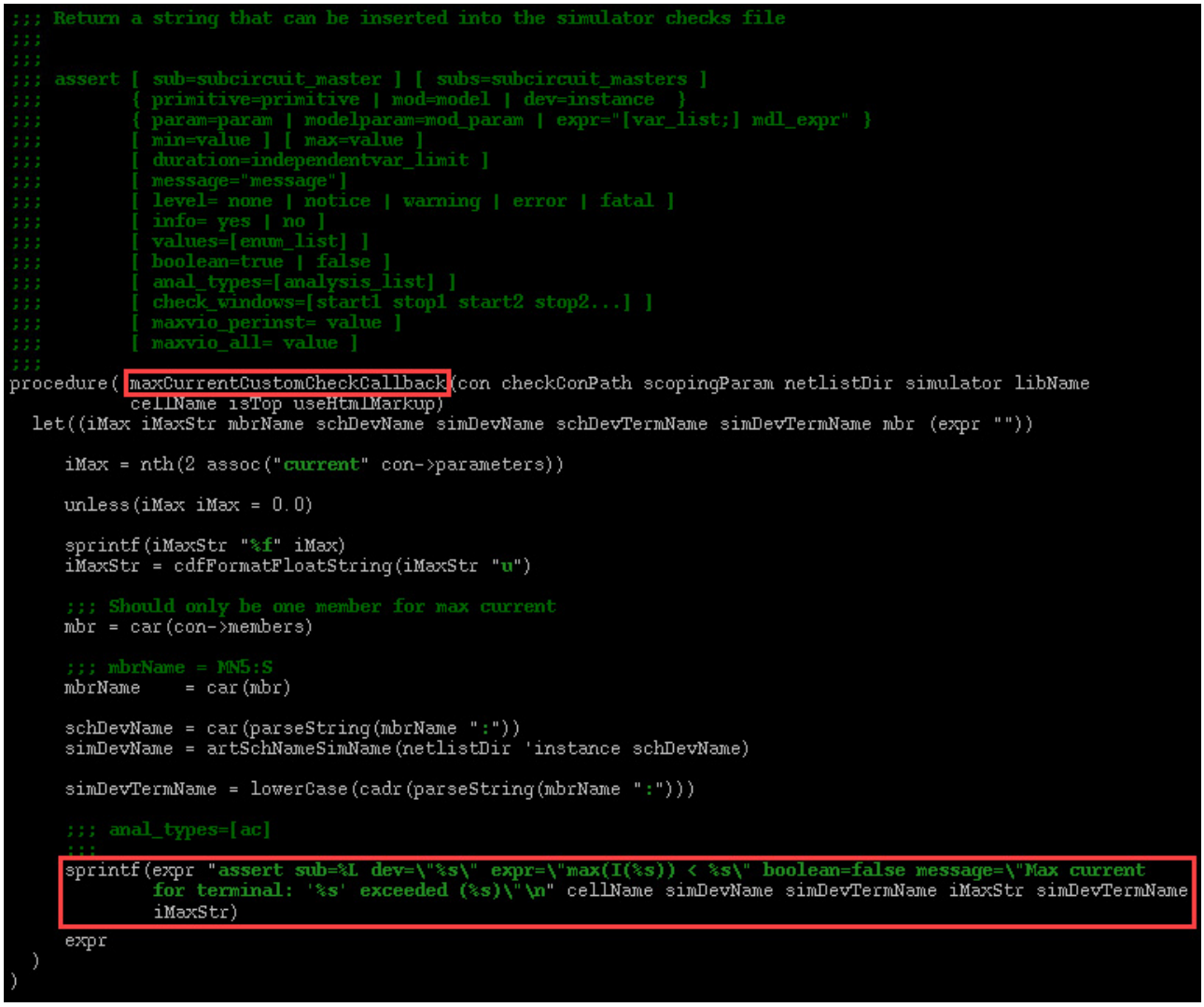
Consequently, when you create a constraint of this constraint type in the Constraint Manager assistant, the same constraint starts to show in the Checks/Asserts assistant too, as shown in the image below.



In addition, you can specify a custom callback, as shown below, in the `config.xml` file to allow the constraints to be netlisted as Spectre asserts along with the existing checks and asserts.

```
<ConstraintType>
   <Name>maxCurrent</Name>
   <GUIName>Max Current</GUIName>

   <!-- customAssertion category -->
   <Categories>frontEnd,physical,customAssertion</Categories>

   <!-- CustomSimulationCheckCB tag -->
   <CustomSimulationCheckCB>maxCurrentCustomCheckCallback</CustomSimulationCheckCB>

   <AllowedMemberTypes>instTerm,pin</AllowedMemberTypes>
   <MinMembers>1</MinMembers>
   <MaxMembers>1</MaxMembers>
```

The following image illustrates the procedure for `maxCurrentCustomCheckCallback` used in the custom callback example above:



```
;;; Return a string that can be inserted into the simulator checks file
;;;
;;;
;;; assert [ sub=subcircuit_master ] [ subs=subcircuit_masters ]
;;;        { primitive=primitive | mod=model | dev=instance  }
;;;        { param=param | modelparam=mod_param | expr="[var_list;] mdl_expr" }
;;;        [ min=value ] [ max=value ]
;;;        [ duration=independentvar_limit ]
;;;        [ message="message"]
;;;        [ level= none | notice | warning | error | fatal ]
;;;        [ info= yes | no ]
;;;        [ values=[enum_list] ]
;;;        [ boolean=true | false ]
;;;        [ anal_types=[analysis_list] ]
;;;        [ check_windows=[start1 stop1 start2 stop2...] ]
;;;        [ maxvio_perinst= value ]
;;;        [ maxvio_all= value ]
;;;
procedure( maxCurrentCustomCheckCallback(con checkConPath scopingParam netlistDir simulator libName
        cellName isTop useHtmlMarkup)
  let((iMax iMaxStr mbrName schDevName simDevName schDevTermName simDevTermName mbr (expr ""))

    iMax = nth(2 assoc("current" con->parameters))

    unless(iMax iMax = 0.0)

    sprintf(iMaxStr "%f" iMax)
    iMaxStr = cdfFormatFloatString(iMaxStr "u")

    ;;; Should only be one member for max current
    mbr = car(con->members)

    ;;; mbrName = MN5:S
    mbrName    = car(mbr)

    schDevName = car(parseString(mbrName ":"))
    simDevName = artSchNameSimName(netlistDir 'instance schDevName)

    simDevTermName = lowerCase(cadr(parseString(mbrName ":")))

    ;;; anal_types=[ac]
    ;;;
    sprintf(expr "assert sub=%L dev=\"%s\" expr=\"max(I(%s)) < %s\" boolean=false message=\"Max current
            for terminal: '%s' exceeded (%s)\"\n" cellName simDevName simDevTermName iMaxStr simDevTermName
            iMaxStr)
    expr
  )
)
```

The arguments for `maxCurrentCustomCheckCallback` are as follows:

con

The custom assert being netlisted. The constraint customization SKILL commands (prefix ci) can be used to get the custom assert name, parameters and members. For example, `ciConGetName(con)`, `ciConListParams(con)`, `ciConListMembers(con)`.

This information can be used to form the simulator assert statement.

t_checkConPath	The simulation path to the custom assert being netlisted. For example, `ether_adc45n\\/ adc_cascode_opamp\\/Constr_1`.
t_scopingParam	This argument is not used.
t_netlistDir	The path to the simulation netlist directory. For example, `simulation/ether_adc45n_sim/ adc_cascode_opamp_sim/adexl/results/data/ Interactive.270/1/ACGainBW/netlist`. The netlistDir can be used to map schematic device/terminal names to simulator device/terminal names. For example, `simDevName = asiMapOutputName(netlistDir 'instance schDevName)`
s_simulator	The simulator used, for example `spectre`. Only the spectre simulator supports custom checks/asserts.
t_libName	The library name. For example, `ether_adc45n`.
t_cellName	The cell name. For example, `adc_cascode_opamp`.
g_isTop	If set to `t`, the top level cellview is netlisted. If set to `nil`, a sub-circuit is being netlisted.
g_useHtmlMarkup	If set to `t` then HTML markup tags are allowed in the string returned by the custom assert netlisting callback. If set to `nil`, the string must only contain plain text.

The following image illustrates how the constraints are netlisted as Spectre assert statements:

```
////////////////////////////////////////////////////
// ether_adc45n.adc_cascode_opamp:constraint
////////////////////////////////////////////////////


////////////////////////////////////////////////////
// Spectre Dynamic Checks
////////////////////////////////////////////////////

ether_adc45n\/adc_cascode_opamp\/Constr_4 dyn_exi inst=[M3] subckt=[adc_cascode_opamp]

////////////////////////////////////////////////////
// Spectre Custom Checks
////////////////////////////////////////////////////

ether_adc45n\/adc_cascode_opamp\/Constr_3 assert sub="adc_cascode_opamp" dev="M5" expr="max(I(s)) < 10.00000u"
                            boolean=false message="Max current for terminal: 's' exceeded (10.00000u)"

ether_adc45n\/adc_cascode_opamp\/Constr_2 assert sub="adc_cascode_opamp" expr="min(V(OUT)) >= 0m && max(V(OUT)) <= 0.001000m"
                            boolean=false message="Voltage for net 'OUT' outside range Min(0m) - Max(0.001000m)"

ether_adc45n\/adc_cascode_opamp\/Constr_2_1 assert sub="adc_cascode_opamp" expr="min(V(OUT)) >= 0m"
                            boolean=false message="Voltage for net 'OUT' below Min(0m)"

ether_adc45n\/adc_cascode_opamp\/Constr_2_2 assert sub="adc_cascode_opamp" expr="max(V(OUT)) <= 0.001000m"
                            boolean=false message="Voltage for net 'OUT' above Max(0.001000m)"

ether_adc45n\/adc_cascode_opamp\/Constr_1 assert sub="adc_cascode_opamp" expr="min(V(OUT)) >= 0m && max(V(OUT)) <= 0.001000m"
                            boolean=false message="Voltage for net 'OUT' outside range Min(0m) - Max(0.001000m)"

ether_adc45n\/adc_cascode_opamp\/Constr_1_1 assert sub="adc_cascode_opamp" expr="min(V(OUT)) >= 0m"
                            boolean=false message="Voltage for net 'OUT' below Min(0m)"

ether_adc45n\/adc_cascode_opamp\/Constr_1_2 assert sub="adc_cascode_opamp" expr="max(V(OUT)) <= 0.001000m"
                            boolean=false message="Voltage for net 'OUT' above Max(0.001000m)"

ether_adc45n\/adc_cascode_opamp\/Constr_0 assert sub="adc_cascode_opamp" expr="min(V(OUT)) >= 0m && max(V(OUT)) <= 1000000.000000m"
                            boolean=false message="Voltage for net 'OUT' outside range Min(0m) - Max(1000000.000000m)"

ether_adc45n\/adc_cascode_opamp\/Constr_0_1 assert sub="adc_cascode_opamp" expr="min(V(OUT)) >= 0m"
                            boolean=false message="Voltage for net 'OUT' below Min(0m)"

ether_adc45n\/adc_cascode_opamp\/Constr_0_2 assert sub="adc_cascode_opamp" expr="max(V(OUT)) <= 1000000.000000m"
                            boolean=false message="Voltage for net 'OUT' above Max(1000000.000000m)"


////////////////////////////////////////////////////
// Spectre APS and XPS Static Checks
////////////////////////////////////////////////////

ether_adc45n\/adc_cascode_opamp\/Constr_5 static_mosv inst=[M2] subckt=[adc_cascode_opamp]
```

Examples of Using the config.xml File

Example 1: Extending an Existing System Constraint Type

The *alignment* constraint is a system constraint type. The user wants to add a new parameter `alignment_custom_param_first` to the alignment constraint. The original `config.xml` already contains an entry for the *alignment* constraint and it might not always be possible to modify it. So, the user can create a new `config.xml` with the following entry:

```
<ConstraintType>
    <Name>alignment</Name>
    <Param>
        <Name>align_custom_param_first</Name>
        <GUIName visibleInEditor="true">AlignmentCustomParamFirst</GUIName>
        <Type>int</Type>
        <ValueRange>0 7 9 -5</ValueRange>
        <DefaultValue>5</DefaultValue>
        <Scope>constraint</Scope>
    </Param>
</ConstraintType>
```

When this `config.xml` is read, there are two children of the `ConstraintType` tag—`Name` and `Param`.

The `Name` tag is anyway a must tag and is present in the in-memory configuration of *alignment* constraint. Therefore, it is ignored. The `align_custom_param_first` parameter is found to be unique and is appended to the *alignment* constraint entry in the in-memory configuration. A dump of the in-memory configuration will show this parameter added to the alignment constraint entry.

Now, assume that the user created another `config.xml` file, with the following entry:

```
<ConstraintType>
    <Name>alignment</Name>
    <Param>
        <Name>align_custom_param_second</Name>
        <GUIName visibleInEditor="true">AlignmentCustomParamSecond</GUIName>
        <Type>string</Type>
        <DefaultValue>My custom string</DefaultValue>
        <Scope>constraint</Scope>
    </Param>
</ConstraintType>
```

When this second `config.xml` file is loaded, the `align_custom_param_second` parameter is also found to be unique and is appended to the *alignment* constraint entry in the in-memory configuration. So, a dump of the in-memory configuration will now show the two parameters, `align_custom_param_first` and `align_custom_param_second`, added to the *alignment* constraint entry.

Example 2: Creating a New Custom Constraint Type

The following is an example of a `config.xml` file containing a new constraint type:

```
<ConstraintType>
    <Name>CstNoiseAndCrosstalk</Name>

    <VerifyCB>CstVerifyCallback</VerifyCB>
    <ValidateCB>CstValidateCB</ValidateCB>

    <Param>
        <Name>CstCrosstalkValue</Name>
        <Type>double</Type>
        <DefaultValue>0.5</DefaultValue>
    </Param>

    <Param>
        <Name>CstMetalFillLayer</Name>
        <Type>enum</Type>
        <DefaultValue>Metal1</DefaultValue>
        <ValueRange>Metal1 Metal2 Metal3 Metal4 Metal5 Metal6 Metal7 Metal8
        Metal9</ValueRange>
    </Param>

    <Param>
        <Name>CstParSameNet</Name>
        <Type>int</Type>
        <DefaultValue>0</DefaultValue>
    </Param>

    <Param>
        <Name>CstReserveLayersForOverBlockRouting</Name>
        <Type>string</Type>
        <DefaultValue>Metal1 Metal2 Metal3 Metal4 Metal5 Metal6 Metal7 Metal8
        Metal9</DefaultValue>
    </Param>
```

```
    <Param>
        <Name>hierarchicalScope</Name>
        <Type>enumset</Type>
        <ValueRange>local below above</ValueRange>
        <DefaultValue>local below</DefaultValue>
        <Scope>constraint</Scope>
    </Param>
</ConstraintType>
```

Here,

- The only parameter `Type` allowed are `int`, `double`, `string`, and `enum`.
- The icon names must be, for example, `CstCrosstalkValue.png` and `CstReserveLayersForOverBlockRouting.png`.

  For more information, see Adding a New Icon to the Constraint Manager Assistant.
- A private prefix and a unique name and type pair is required for each parameter.

  The same constraint name cannot be reused as it cause additional parameters to be added to that existing constraint. For example, `Cst` has been used as the prefix in the example above for “customer”.

  When you define a `ConstraintType` named `symmetry`, parameters are added to the symmetry constraint, if required.

  Virtuoso applications will only recognize pre-defined constraints and their pre-defined parameters, and not the extended set.

  Note: Pre-defined constraint and parameter names have a reserved name space, that being any name starting with an alphabetic and lower-case character.

  See also Using Validation and Verification Callbacks.
- Parameter names cannot be reused unless their definitions match perfectly. It is therefore not recommended that you reuse parameters names as it may lead to errors.

Example 3: Overriding an Existing Custom Constraint Type

Following are entries for a custom constraint type, `custom_conType`, in two different `config.xml` files:

First config.xml file:

```
<ConstraintType>
```

```
    <Name>custom_conType</Name>
    <GUIName >CustomConType</GUIName>
    <Param>
        <Name>customParam</Name>
        <GUIName visibleInEditor="true">CustomParam</GUIName>
        <Type>int</Type>
        <ValueRange>0 7 9 -5</ValueRange>
        <DefaultValue>5</DefaultValue>
        <Scope>constraint</Scope>
    </Param>
</ConstraintType>
```

Second config.xml file:

```
<ConstraintType>
    <Name>custom_conType</Name>
    <GUIName >Custom Con Type</GUIName>
    <Param>
        <Name>customParam1</Name>
        <GUIName visibleInEditor="true">CustomParam</GUIName>
        <Type>int</Type>
        <ValueRange>0 7 9 -5</ValueRange>
        <DefaultValue>5</DefaultValue>
        <Scope>constraint</Scope>
    </Param>
    <Param>
        <Name>customParam2</Name>
        <GUIName visibleInEditor="true">CustomParam</GUIName>
        <Type>string</Type>
        <DefaultValue>paramVal2</DefaultValue>
        <Scope>constraint</Scope>
    </Param>
    <VerifyCB>myVerifyCB</VerifyCB>
</ConstraintType>
```

Here, first one is the closer `config.xml` (found earlier) and the second one is a distant one. When first one is loaded, a new constraint of the given type is created with the specified GUI name and a parameter, `customParam`. When the second one is loaded, the constraint type defined in the previous one is completely overridden with the new definition. Now, the constraint type has a new GUI name and two parameters, `customParam1` and `customParam2` along with a verification procedure.

2

Customizing the Circuit Prospector

Note: The *Circuit Prospector* recognizes structures using place-holder names for device, parameter, terminal and net names. As each Process Design Kit (PDK) and design library can use any names, each PDK and library must register the names used by each object for the respective place-holder names. For more information, see PDK Setup for the Circuit Prospector.

The *Circuit Prospector* assistant can be customized to identify circuit patterns in a schematic, list the found members, as per each searched pattern, as candidates for constraint setting, and then suggest a set of relevant constraints.

Important

If you have a requirement for more extensive examples of *Circuit Prospector* customization, above those included in this chapter, you should contact your local account manager or Cadence support directly for more information on how to access this information.

You can then selectively choose a subset of candidates and apply the associated constraints (using the *Create Default Constraints* option in the *Constraint Manager* assistant).

Each particular circuit pattern to identify, and the associated constraints, are described in *finders*, with subsets of finders being grouped into *categories*.

This chapter describes how to create and register customized constraint generators, new finders, new categories, and how to load them into the *Circuit Prospector*.

The following topics are discussed:

- Creating Custom Constraint Generators
- Creating New Finders
- Creating New Categories

Important

For information on removing any PDK dependencies from the *Circuit Prospector*, see the relevant SKILL API commands in the Circuit Prospector Assistant Customization SKILL Commands section of the *Virtuoso Unified Custom Constraints SKILL Reference*.

PDK Setup for the Circuit Prospector

To set up a PDK with customization for the *Circuit Prospector*, use the `libInit.il` file to define the components for recognition by the *Circuit Prospector*.

Note: The `libInit.il` file is a reserved file name available in each library for automatic loading.

For assistance, a setup example for `gpdk090` is detailed below:

```
when( isCallable('ciRegisterDevice)
ciRegisterDevice("nfet" append(ciGetDeviceNames("nfet") '(
'("gpdk090" "nmos1v" nil)
'("gpdk090" "nmos1v_hvt" nil)
'("gpdk090" "nmos1v_iso" nil)
'("gpdk090" "nmos1v_nat" nil)
'("gpdk090" "nmos2v" nil)
'("gpdk090" "nmos2v_nat" nil)
)))
ciRegisterDevice("pfet" append(ciGetDeviceNames("pfet") '(
'("gpdk090" "pmos1v" nil)
'("gpdk090" "pmos1v_hvt" nil)
'("gpdk090" "pmos2v" nil)
)))
ciRegisterDevice("npn" append(ciGetDeviceNames("npn") '(
'("gpdk090" "npn" nil)
)))
ciRegisterDevice("pnp" append(ciGetDeviceNames("pnp") '(
'("gpdk090" "pnp" nil)
'("gpdk090" "vpnp2" nil)
'("gpdk090" "vpnp5" nil)
'("gpdk090" "vpnp10" nil)
)))
ciRegisterDevice("capacitor" append(ciGetDeviceNames("capacitor") '(
'("gpdk090" "mimcap" nil)
)))
ciRegisterDevice("resistor" append(ciGetDeviceNames("resistor") '(
'("gpdk090" "resm1" nil)
'("gpdk090" "resm2" nil)
'("gpdk090" "resm3" nil)
'("gpdk090" "resm4" nil)
'("gpdk090" "resm6" nil)
'("gpdk090" "resm6" nil)
'("gpdk090" "resm7" nil)
'("gpdk090" "resm8" nil)
'("gpdk090" "resm9" nil)
'("gpdk090" "resnsndiff" nil)
'("gpdk090" "resnspoly" nil)
'("gpdk090" "resnspdiff" nil)
'("gpdk090" "resnsppoly" nil)
'("gpdk090" "resnwoxide" nil)
'("gpdk090" "resnwsti" nil)
'("gpdk090" "ressndiff" nil)
'("gpdk090" "ressnpoly" nil)
'("gpdk090" "resspdiff" nil)
'("gpdk090" "ressppoly" nil)
)))
```

```
ciRegisterDevice("fet" append(ciGetDeviceNames("nfet") ciGetDeviceNames("pfet")))
ciRegisterDevice("bjt" append(ciGetDeviceNames("npn") ciGetDeviceNames("pnp")))
ciRegisterDevice("passive" append( ciGetDeviceNames("capacitor")
ciGetDeviceNames("resistor")))
ciRegisterDevice("passive" append( ciGetDeviceNames("passive")
ciGetDeviceNames("inductor")))

) ; when
when( isCallable('ciMapParam)
ciMapParam("width" append(ciGetParamMapping("width") '("w")))
ciMapParam("length" append(ciGetParamMapping("length") '("l")))
ciMapParam("fingerCount" append(ciGetParamMapping("fingerCount") '("fingers")))
ciMapParam("fingerWidth" append(ciGetParamMapping("fingerWidth") '("fw")))
ciMapParam("mFactor" append(ciGetParamMapping("mFactor") '("m")))

) ; when
when( isCallable('ciMapTerm)
ciMapTerm("gate" append(ciGetTermNames("gate") '("G")))
ciMapTerm("source" append(ciGetTermNames("source") '("S")))
ciMapTerm("drain" append(ciGetTermNames("drain") '("D")))
ciMapTerm("bulk" append(ciGetTermNames("bulk") '("B")))
) ; when
```

Creating Custom Constraint Generators

A constraint generator is a SKILL expression that, when evaluated, creates a constraints. The expression is evaluated within an internal function that makes available variables for accessing the selected instances, nets, and pins (`instsNetsPins`) and the constraints cache. You can create and register customized constraints using the `ciRegisterConstraintGenerator` SKILL function. When registering your own plug-in constraints, you can choose to optionally add them to the *Constraint Manager* toolbar and/or the *Generate Constraints* constraint list in the *Constraint Manager* context-menu. The constraint generator can also be set as the default generator for a *Circuit Prospector* finder.

Registering a Custom Constraint Generator

When you use the `ciRegisterConstraintGenerator()` SKILL function, it registers the specified constraint generator with the *Constraint Manager* assistant and adds an icon for it to the *Constraint Manager* toolbar. This means that the new constraint generator can be used to generate one or more constraints with preset parameters from the selections made on the canvas.

The custom constraint generator can be registered by loading the SKILL file into the application or by storing the file in a `.cadence/dfII/ci/generators` directory prior to launching a DFII session.

Note:

- The SKILL file must have a `.il` extension.
- The `.cadence` directory must be on the file search path.

For information about how to load files with custom constraint generator after Virtuoso startup, refer to Loading Customized Startup Files On Demand.

The settings of a constraint generator dialog box are stored in the `.cadence/dfII/ci/1/<generatorName>_cGen.ini` file. When you call the dialog box next time, it is opened with the previous settings.

When you select multiple devices for constraint creation and open any constraint generator dialog box, a button named *OK All* is also displayed in the dialog box to apply the same settings for the entire selection. However, some settings are device specific and might not be applicable to all device selections. In this case, the default settings are used.

See also:

- Creating and Overriding Custom Constraint Types

- Constraints SKILL API Commands in the *Virtuoso Unified Custom Constraints SKILL Reference*

The constraint generator can optionally specify arguments that might display a window when the generator is run. This is to allow you to specify values for these arguments. The values of the arguments may then determine what constraints are generated and their parameter settings. The argument values are made available to the constraint generator expression through the variable, `args`. This variable is a disembodied property list and the argument values can be accessed through the argument name, for example `args->strength`.

The argument types can be one of the following: `bool`, `int`, `float`, `enum`, `string` (a single-line string), `multiString` (a multi-line string), `pattern` (a multi-line string with fixed-pitch font), `orient` (a multi-line string with fixed-pitch font), `separator`, `beginExpandedOptions`, or `endExpandedOptions`. The `pattern`, `orient`, `separator`, `beginExpandedOptions`, and `endExpandedOptions` argument types are described in more detail in the *Virtuoso Unified Custom Constraints SKILL Reference*.

A drop-down list box can be added to the constraint generator dialog box by declaring it using one of the following ways:

- Set the type to `string` and the widget type to `"comboBox"`.

  The valid values can contain spaces and can be updated dynamically in the callback. To add new valid values in the callback, use one of the following ways:

  - Set the argument to a value that does not already belong to the valid values list.
  - Update the valid values through the widget properties, as shown below:

```
("dyn_cb" string "\"value 2\"" widgetType  "comboBox" legalValues "\"value
1; value 2; value 3; value 3\"")
    -> dynamic combo box allowing spaces in name. 'value 2' will be the
default
```

- Set the type to `enum`.

  The valid values can contain spaces, but cannot be updated automatically in the constraint generator dialog box. The format for defining valid values with spaces and modifiable drop-down list box is illustrated below:

```
("enum_cb" enum "" legalValues "\"value 1; value 2; value 3; value 2\"")
    -> non-dynamic combo box allowing spaces in name. 'value 2' will be the
default
("enum_cb1" enum "\"value 1\"" legalValues "\"value 0; value 1; value 2; value
3\"")
    -> non-dynamic combo box allowing spaces in name. 'value 1' will be the
default
```

The example below shows how to register the *Matching (strength)* constraint generator that is displayed as an option in the *Constraint Manager* toolbar and the *Generate Constraints* context-menu:

```
ciRegisterConstraintGenerator(
    list(nil
        'name "Matching (strength)"
        'description "Generate various levels of Matching constraints"
        'expression "ciRunMatchingConstraintsGenerator(args instsNetsPins cache)"
        ;;; expression to generate constraints
        'addToToolbar t ;;; whether you want to add a button to the toolbar for
        this generator
        'iconName "templateMatched" ;;; icon to use on the toolbar.
        templateMatched.png must exist in the icon search path
        'args list( "strength" 'enum "low" "medium" "high")
        'menu list( "Custom1" "Custom2")
        )
)
```

Note: The `menu` argument is optional.

Constraint Template Storage

Template storage is possible for all constraints that have independent Open Access (OA) storage.

However, the fixed, locked, and net priority constraints do not have independent storage, and cannot be stored in templates as they are OA object attributes (although they are present in the *Constraint Manager* as constraints).

If you attempt to add one of these constraint types into a template, the *Constraint Manager* will issue a warning and will not add that constraint to template storage. The constraint will still remain visible in the *Constraint Manager*, but will not be presented and stored as part of a template.

Setting Widget Properties in Constraint Generator Arguments

While defining the constraint generator arguments, the widget properties can be set by adding a `'widgetProperties` directive to the argument definition. This directive accepts one of the following inputs:

- A list of widget property list
- An expression returning a list of widget property list

In the constraint generator argument, the list of widget property can have two to three elements. If there are three elements, the list is of the following format:

```
list('widgetName 'propName propNameValue)
```

The value can also be set as a string expression. To set a value of string type, ensure that the value is wrapped within double quotes. For example: `value = "\"sampletext"\"`

If there are two elements in the property list, the `'widgetName` directive is automatically set to the `argName`.

For example:

```
procedure(myTab1Props()
    list(
        list('title "\"Tab 1\"")
        list( 'toolTip"\"some explanation1\"")
    )
)
Arg1 = list("Tab1" 'separator 'widgetType "tabSeparator"
        'widgetProperties "myTab1Props()")
```

Setting Properties in Callback Procedure

The properties of all widgets are appended at the end of the `args` function in the callback procedure. To access the widget properties, use one of the following syntaxes:

- `properties = args->widgetProperties`
- `ciGetWidgetProperties(args)`

The format of the properties is a disembodied property list of the following format:

```
list('widgetProperties
    'widgetName1 list('propertyList'propName1 value1 … 'propNameN valueN)
    …
    'widgetNameX list('propertyList'propNamei valuei… 'propNameX valueX)
)
```

Consequently, a property can be set in the following way:

```
properties->widgetName->propName= myValue
```

Note: String values are not considered as expressions.

Accessing a Widget Name

Widget names are constructed based on the argument names set for a generator. While doing so, the following should be kept into consideration:

- The argument and the widget used to edit its value should have the same name.
- The name of the label in front of the argument name is `argName.label`.
- The name of any widget created by using `legalValues` on the argument is of the format `argName.legalValue1`.

For example:

```
list("CheckBoxes" 'string "" 'widgetType "checkBoxGroup"
    'legalValues "\"cb1; cb2; cb3\"")
```

Creates the following widgets on which properties can be set:

- `CheckBoxes`
- `CheckBoxes.label`
- `CheckBoxes.cb1`
- `CheckBoxes.cb2`
- `CheckBoxes.cb2`

Available Widget Properties

You can use the following properties for your widgets:

- `toolTip`
- `text` (Not for `textEditwidget` or `groupBoxes`)
- `title` (Equivalent to `text` for group boxes and tabs)
- `hide`
- `enable`
- `guiOnly`

These properties support the following types of values:

- A string for `toolTip`, `text`, and `title`
- A Boolean for `hide`, `enable`, and `guiOnly`

However, remember that the widget properties do not update the status of the parameter dynamically in the Constraint Manager. In addition, changing the text of a label on the argument does not update the parameter name in the Constraint Manager. The name can be changed in the `config.xml` file.

Enabling or Disabling Callbacks to Widget Properties

To enable or disable the triggers to callbacks for the status of widget properties, set `'widgetPropertiesEnabled` with the `'settings` argument of the `ciRegisterConstraintGenerator` SKILL function. For example,

```
ciRegisterConstraintGenerator(list(nil
    'name           "MyTemplateForWarnings"
    'description    "Callback will create warnings"
    'expression     "ciTemplateCreate(cache \"MyTemplateForWarnings\" ?members
    instsNetsPins)"
    'addToToolbar   t
    'iconName       "CurrentMirror"
    'args           "ciGetStructArgs('MyTemplateForWarnings)"
    'menu           "Rapid Analog Prototype"
    'useCGenForEdit t
    'templateName   "MyTemplateForWarnings"
    'settings        list(nil 'widgetPropertiesEnabled nil)
    )
)
```

By default, triggers to callbacks for widget properties are disabled, unless some `'widgetProperties` are defined on any argument. To enable such triggers, set `'widgetPropertiesEnabled` to `t`. The following table shows the status of `'widgetProperties` in the different scenarios:

	`'widgetProperties` defined on generator arguments	**`'widgetProperties` not defined on generator arguments**
`widgetPropertiesEnabled == t`	Enabled	Enabled
`widgetPropertiesEnabled == nil`	Disabled	Disabled
`widgetPropertiesEnabled` are not set on the generator definition	Enabled	Disabled

Note: When widget properties are disabled, then in the callback, `args->widgetProperties` returns `disabled`.

Viewing Arguments in GUI Only and Not Saving as Template Parameters

By default, arguments are stored as template parameters. You can control this behavior using the `guiOnly` widget property. This property can accept `t` or `nil` as a value, where `nil` is the default.

To avoid storing of arguments as template parameters, set the `guiOnly` widget property to `t`.

However, the arguments of following types are always `guiOnly` and cannot be set to be saved as template parameters:

- `beginExpander`
- `endExpander`
- `separator`
- `tabSeparator`

Note: The value of the argument for which `guiOnly` widget property is set to `t` is not saved in a template. If the value needs to be retrieved when loading a constraint generator dialog box to edit the template, it is up to you to handle how to store and how to retrieve the value.

For example, any template generator registered with the following arguments will have only two parameters `'area'` and `'perimeter'`:

```
myArgs = list(  list( "form" 'string "\"Square\"" 'widgetType "radioButtons"
'legalValues "\"Square; Rectangle; Circle\""
    'widgetProperties '((guiOnly t) (form\.label text "\"Select form:\"")) 
    'callback callback ) ;; not saved as a parameter
    list("r0" 'float "float(1 + 1)" 'callback callback 'widgetProperties '((guiOnly
    t) (r0\.label text "\"Edge [m]\""))) ;; not saved as a parameter
    list("r1" 'float 0.0 'callback callback 'widgetProperties '((guiOnly t) (hide
    t) (r1\.label hide t))) ;; not saved as a parameter
    list("area" 'float 0.0 'callback callback 'widgetProperties '((area\.label
    text "\"Area [m^2]\""))    'readOnly) ;template parameter
    list("perimeter" 'float 0.0 'callback callback 'widgetProperties
    '((perimeter\.label text "\"Perimeter [m]\"")) 'readOnly) ;template parameter
)
```

Overriding the Behavior of Defaults Button in Constraint Generator Forms

On the various constraint generator forms, the *Defaults* button can be used to rollback any user-defined values in the fields to their valid default values. This means that the button resets

the fields to their default values after the callbacks have been triggered. This is the system-provided behavior of the *Defaults* button. However, you can override this behavior by setting `defaultButtonCallback` for the constraint generator to the name of the procedure that needs to be called when you click the *Defaults* button. The following arguments are supported in this callback:

- `cache`
- `instsNetsPins`
- `oldArgs` (First valid arguments)
- `defaultArgs` (Original default before the saved values (from template or settings) are loaded and callbacks have not yet been triggered.)
- `args` (Current arguments in the constraint generator dialog box)

The following syntax shows the usage of these arguments:

```
ciRegisterConstraintGenerator(list(nil
    'name          "MyGenericModgen" ;; Must match the template name which is the
    structure symbol name as a string
    ...
    ...
    'settings '(nil widgetPropertiesEnabled t helpTag "genericModgenRAP"
    'defaultButtonCallback "defaultButtonCallback(cache instsNetsPins oldArgs
    defaultArgs args)")
    )
)
```

Example of Setting Widget Properties

Generator Arguments

```
myExampleArgs= list(
    list("myGroupBox" 'separator);; Regroup widget into a groupBox
    list( "RadioButtons" ;; Argument Name
        'string"\"Show Arg1 + label\"" ;;buttons selected by default
        'widgetType "radioButtons" ;;A group box of radio buttons
        'legalValues "\"Show All 'GroupBox Title'; Hide Field 'GroupBox Title';
        Show Field 'GroupBox Title'; Hide Label 'GroupBox Title'; Show Label
        'GroupBox Title'; Hide All 'GroupBox Title'\"" ;; Four radio buttons
        'callback "myExampleCallback(
            cache argName args oldArgs instsNetsPins userEdit)"
            ;; callback with all available argument
```

```
        'widgetPropertieslist(
            list('enable nil )
                ;;disable radio buttons
            list('RadioButtons\.label 'text "\"NewRadiobuttonsText\""
                ;;update the text for the radiobuttons
            list('Radiobuttons\.Show\All\'GroupBox\Title
                'tooltip "tooltipExpression()"
            ;;The tooltip set on the radio buttons named Show, All, 'GroupBox,
            ;;Title will be set to the string value returned by the procedure
            ;;tooltipExpression
    )
)
...
...
list("GroupBox Title" ;;Argument group box title
    'string "\"\"" ;; A line edit
    'callback "myExampleCallback(
        cache argName args oldArgs instsNetsPins userEdit)"
        ;;Note: callback does not need to be the same
    )
list("Change label"
    'string "\"\"" ;;A line edit
    'callback "myExampleCallback(
        cache argName args oldArgs instsNetsPins userEdit)"
        ;;Note: callback does not need to be the same
    )
)
```

Generator Definition

```
ciSetStructArgs('MyExamplemyExampleArgs)
ciSetStructGeneratorExpressions('MyExample ciGetStructGeneratorExpressions(
        'GenericModgen))
procedure(createMyCon(cache instsNetsPins structType args @key (params nil)
        (userParamsnil))
    let( (templateCons)
        templateCons= ciConCreateExpanded(cache 'modgen ?members instsNetsPins)
    )
)
ciRegisterConstraintGenerator(
    list(nil 'name "MyExample"
```

```
        'description "An example on how to change properties on widgets"
        'expression "ciTemplateCreate(cache \"MyExample\" ?members instsNetsPins)"
        'addToToolbar t
        'iconName "CurrentMirror"
        'args "ciGetStructArgs('MyExample)"
        'menu "Rapid Analog Prototype"
        'useCGenForEdit t
        'templateName "MyExample"
        'settings list(nil 'widgetPropertiesEnabled t)
        ;;The 'setting argument is not mandatory in this example because widget
        ;;properties have been defined on some arguments of the generator.
    )
)
ciTemplateCreateDefinition("MyExample" 'MyExample 'createMyCon
        ?acceptsUserParams t)
ciReinitStructTemplateDefs()
```

Generator Callback

```
procedure(myExampleCallback(cache argName args oldArgs instsNetsPins userEdit)
let(((properties args->properties) (lbl nil))
    case(argName ;;argName is the argument that triggers the callback
    ("RadioButtons"
        when(isRadioButtonCheck(args "Show All 'GroupBox Title'")
            properties->GroupBox\Title->hide = nil
            properties->GroupBox\Title\.label->hide = nil
        )
        when(isRadioButtonCheck(args "Hide Field 'GroupBox Title'")
            properties->GroupBox\Title->hide = t )
        when(isRadioButtonCheck(args "Show Field 'GroupBox Title'")
            properties->GroupBox\Title->hide = nil )
        when(isRadioButtonCheck(args "Hide Label 'GroupBox Title'")
            properties->GroupBox\Title\.label->hide = t)
        when(isRadioButtonCheck(args "Show Label 'GroupBox Title'")
            properties->GroupBox\Title\.label->hide = nil )
        when(isRadioButtonCheck(args "Hide All 'GroupBox Title'")
            properties->GroupBox\Title->hide = t
            properties->GroupBox\Title\.label->hide = t
        )
    )
    ("GroupBox Title" properties->myGroupBox->title = args->GroupBox\Title )
```

```
    ("Change label"
        lbl= args->Change\label
        when(lbl&& lbl!= "" properties->GroupBox\Title\.label->text = lbl)
    )
    )
    args
);;end let
);; end procedure
;;Procedure to check if a radio button is checked
procedure(isRadioButtonCheck(args radioButton)
    exists(x parseString(args->RadioButtons";") (x == radioButton))
)
```

Creating New Iterators

You can create and register customized iterators using a SKILL API.

After registering your own plug-in and iterators using `ciRegisterIterator`, you will then be able to select them from the *Search Using Iterator* drop-down list of iterators in the Edit Finder form. This means that your new iterator can be used by any new finder.

The custom iterator can be registered by loading the SKILL file into the current session or by storing the file in the `.cadence/dfII/ci/iterators` directory. For information about how to load files with custom iterators after Virtuoso startup, refer to Loading Customized Startup Files On Demand.

The matching expression associated with the finder is evaluated by the iterator at run time.

- The SKILL file must have an `.il` extension.
- The `.cadence` directory must be in the file search path (see The Cadence Setup Search File: setup.loc in the *Cadence Application Infrastructure User Guide*).

Creating New Finders

Finders are grouped by *categories* and listed in the *Circuit Prospector* under the *Search for* pull-down list.

When a filter is selected and run, any searched candidates found in the current schematic will be listed in the table in the lower section of the *Circuit Prospector*.

A finder comprises of three elements:

1. An iterator

   An iterator is a pre-filter that retrieves the element found in the schematic on a per type basis (devices, nets or pins), or by symmetry or structure.

2. A matching expression

   A matching expression is used to operate a second level of filtering. The expression is submitted as a string of characters to the iterator for evaluation. If the result of the evaluation for each object of the same sublist is `nil`, then that object is filtered out by the iterator.

3. A default constraint generator

   A default constraint generator defines the set of constraints to apply to the found members of the corresponding searched pattern.

You can create and register customized finders using a SKILL API. When registering your own plug-in finder using the `ciRegisterFinder` function (see Creating a New Finder Using SKILL API), the finder name is added to the current category and to the finder list under the *Search for* pull-down.

When you select the new finder in the *Circuit Prospector*, the list of members of the searched pattern for the corresponding finder will be displayed in the table.

The iterator and constraint generator listed in the `ciRegisterFinder` function must be correctly registered (see Creating Custom Constraint Generators). The custom finder can be registered by loading the SKILL file into the application or by storing the file in a `.cadence/dfII/ci/finders` directory.

Note:

- The SKILL file must have the naming format of `<finder>.il`.
- Where `<finder>` must be the name used to register the finder with an "_" (underscore) character replacing each blank.

- The `.cadence` directory must be on the file search path (see The Cadence Setup Search File: setup.loc in the *Cadence Application Infrastructure User Guide*).

You can create and register customized finders using either of the following methods:

- Using the Edit Finder form (see Creating a New Finder Using the Edit Finder Form)
- Using SKILL API (see Creating a New Finder Using SKILL API)

For information about how to load files with customized finders after Virtuoso startup, refer to Loading Customized Startup Files On Demand.

Creating a New Finder Using the Edit Finder Form

Using the Edit Finder form is simplest way to create a new finder.

You can access the Edit Finder form in the *Circuit Prospector* by clicking on the "*...*" button that is adjacent to the *Search for* field (having already selected a finder from the pull-down).

Note: See also Creating Finders and Editing Finders in the *Virtuoso Unified Custom Constraints User Guide*.

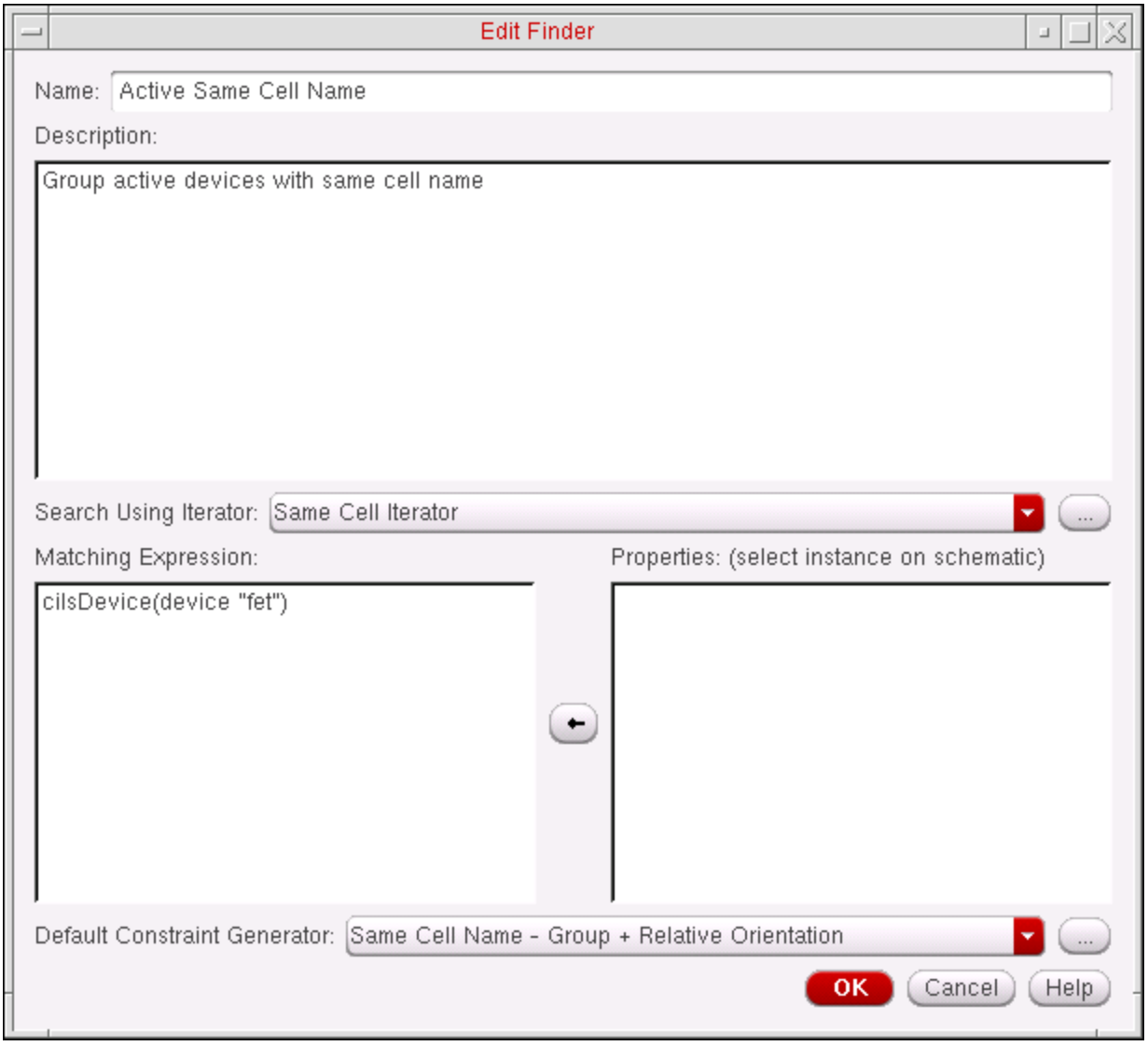


Figure 2-1 The Edit Finder Form

In the Edit Finder form you can:

- Enter a new finder name in the *Name* field.
- Change the *Description* to fit the new finder.
- Select a different iterator from the *Search Using Iterator* pull-down list.
- Modify or enter a new matching expression in the *Matching Expression* section.
- Select an alternative *Default Constraint Generator*.

After you select *OK* to register your changes this will save your new finder as a SKILL file in the current `.cadence/dfII/ci/finders` directory. This new finder will then be listed at the end of the category currently selected in the *Circuit Prospector*, with the changed category being saved in the `.cadence/dfII/ci/categories` directory.

Instead of modifying a category it is preferred that you create a new one (see Creating New Categories).

Note: If you did not enter a new name for the finder before selecting *OK* in the Edit Finder form this will overwrite the current finder. If this was not done intentionally you can remove the corresponding SKILL file from the `.cadence/dfII/ci/finders` directory and start a new DFII session to retrieve the original finder.

Creating a New Finder Using SKILL API

The `ciRegisterFinder` SKILL command can be used to register a finder within the *Circuit Prospector*. The name of the new finder will be added to the end of the current finder list.

Note: Although not recommended, you can modify pre-defined finders using the existing finder name in the `ciRegisterFinder` function. However, by doing so you may miss updates developed in future releases. It is preferred that you create a new finder by adding its name to the list of finders in one or more of your pre-defined or new categories (Creating New Categories).

- Each finder has a name defined as a string under `'name`.
- A comment can be registered as a string under `'description`.
- You can specify the iterator for your finder as a string under `'iterator`.
- The matching expression is defined as a SKILL expression under `'expression`.
- You can store the string name of an existing constraint generator using `'defaultCGen`.

For the matching expression, depending on the selected iterator, you can use any of the following variables and SKILL API functions:

Iterator	**Variable**	**SKILL API**
Same Cell Iterator	device	ciIsNet ciIsDevice ciCreateRoutePriorityCon ciNetOnTerm
Pin Iterator	pin	
Net Iterator	net	ciIsNet
XY Symmetric Iterator	L, R	

For example, to catch all nets that are named with either one of the substrings "clk" or "clock" in your schematic, you can define a new finder in a Clocks_Nets.il file. This file will be stored in the .cadence/dfII/ci/finders directory located in the file search path (see The Cadence Setup Search File: setup.loc in the *Cadence Application Infrastructure User Guide*).

The content of the Clocks_Nets.il file may look like:

```
ciRegisterFinder(
    list(nil
        'name "Clock Nets"
        'description "Find all clock nets by scanning their names"
        'iterator "Net Iterator"
        'expression "if(index(net->name \"clk\") || index(net->name \"clock\")
        then net->name else nil)"
        'defaultCGen "Custom Clock Net Priority"
        'legal CGens nil
    )
)
```

Creating New Categories

A category (also referred to as an *assistant*) is a sequential list of finders. Customized categories can be created and registered using specific Constraints SKILL API Commands.

After registering a customized (plug-in) category using the `ciRegisterAssistant` function (see the Category Creation Example below), the category name will be added to the *Category* pull-down list in the *Circuit Prospector*. When you select this new category, the list of finders for the corresponding category will be displayed in the *Search for* pull-down list.

Each finder listed in the `ciRegisterAssistant` function must be correctly registered for the corresponding category to be successfully registered (see Creating New Finders). The custom category can be registered by loading the associated SKILL file directly into the application or by storing the file in a `.cadence/dfII/ci/categories` directory. The `.cadence` directory must be on the file search path. For more information, see The Cadence Setup Search File: setup.loc in the *Cadence Application Infrastructure User Guide*.

Note: The SKILL file must also be in the format `<categoryName>.il`. Where `<categoryName>` must be the name used to register the category with an "_" (underscore) character to replacing each blank.

For information about how to load files with custom categories after Virtuoso startup, refer to Loading Customized Startup Files On Demand.

Category Creation Example

The `ciRegisterAssistant` function is used to register a category (assistant) in the *Circuit Prospector*, with the name of the new category being added to the end of the *Category* pull-down list.

The current pre-defined categories are *Active Devices*, *Passive Devices*, *Structures*, *Nets*, and *Pins*. For more information see Constraint Categories in the *Virtuoso Unified Custom Constraints User Guide*.

Note: Although not recommended you can modify pre-defined categories using the existing category name in the `ciRegisterAssistant` function. If you do this however you may miss updates to the category that could be made in future releases. It is therefore recommended instead that you create a new category for your custom finders and include as many pre-defined finders as you require in that custom category.

In the *Circuit Prospector*, *Select A Finder* and *ALL* are automatically added to the default list of finders.

Note: The order of the finders is important.

When you select *ALL*, from the *Search for* pull-down list, an internal procedure scans the current schematic and lists all found groups in the order of the finders listed here. When you then select all the found groups and apply the default constraints (by selecting *Create Default Constraints* from the constraints pull-down in the *Constraint Manager*), the corresponding constraints will be set in the order of the selected groups in the *Circuit Prospector*.

For example:

To set constraints versus net types, you should first of all create a list of finders for the *Circuit Prospector* to look for relevant candidates. Then, you can register a custom category for these new finders with, for example, a "`Various_Nets.il`" file containing the following:

```
ciRegisterAssistant(
    list(nil
        'name "Various Nets"
        'description "Net finders"
        'finderNames list(
              "Supply Nets"
              "Ground Nets"
              "Clock Nets"
              "Sample and Hold Nets"
              "Differential Nets"
              "All Nets"
        )
    )
)
```

- Each category has a name defined as a string under `'name`.
- A comment can be registered as a string under `'description`.
- A category is a list of finders to make them accessible in the *Circuit Prospector*, defined as a string list under `'finderNames`.

Loading Customized Startup Files On Demand

On Virtuoso startup, the following type of files are loaded:

- `.cadence/icons/16x16/*.png`
- `.cadence/dfII/ci/config.xml`
- `.cadence/dfII/ci/iterators/*.il *.ile *.cxt`
- `.cadence/dfII/ci/generators/*.il *.ile *.cxt`
- `.cadence/dfII/ci/structures/*.il *.ile *.cxt`
- `.cadence/dfII/ci/finders/*.il *.ile *.cxt`
- `.cadence/dfII/ci/categories/*.il *.ile *.cxt`
- `.cadence/dfII/ci/categories/org/*.il *.ile *.cxt`

All `config.xml` files and the icon files in `.cadence/icons/16x16` directories on the Cadence Search Path are loaded and merged in the order they are found in the `setup.loc` file. If you want to load a customized `config.xml` file on demand after Virtuoso startup, for example when a design library or PDK is loaded, you can use the `ciLoadConfigXML` SKILL function. In the process, the new entries get merged into the existing `config.xml` files that have already been loaded and the existing entries are overwritten. Similarly, you can load specific or all icon files with `.png` extension on demand using the `ciLoadIcon` or `ciLoadIcons` SKILL function, respectively.

To load on demand all constraint-related files within the hierarchy of the specified `.cadence` directory, you can call the `ciLoadDotCadenceFiles` SKILL function from the CIW or from within any other SKILL function. It can also be called within the `libInit.il` file of a PDK or design library so that library-specific constraint customizations can be setup when the library is loaded. As you can have your own customizations, it is recommended that the following command is run to re-load those customizations:

```
ciLoadDotCadenceFiles("./.cadence")
```

3

Editing Constraints Using External Editors

By default, constraints can be edited within the *Constraint Manager* assistant. However, for some specific types of constraints, you can also use an external constraint editor.

The external constraint editors available for use can be categorized as follows:

- Cadence-provided external constraint editors that are available from the *Constraint Manager* assistant, that is, *Cell Planner*, *Module Generator*, and *Process Rule Editor*.

  See Using Cadence-Provided External Constraint Editors.

- Registered (user-defined) custom constraint editors.

  See Using a Custom Constraint Editor.

This chapter describes how to use such external constraint editors for editing constraints.

You can also configure using SKILL functions to use a preferred external constraint editor for editing a constraint template. See Configuring to Open Constraint Editor on Double-Click.

Using Cadence-Provided External Constraint Editors

To access an external constraint editor, select the appropriate editor from the *Constraint Manager* toolbar.

When you have an external constraint editor open for editing a constraint, constraint editing within the *Constraint Manager* is disabled until you close the external constraint editor. At this time, the external constraint editor directly interacts with the constraints database and the *Constraint Manager* assistant keeps updating itself using the constraints database observer mechanism.

For more information, see:

- *The Cell Planner and Module Generator Editor Options* in the *Virtuoso Unified Custom Constraints User Guide*.
- *Editing Modgens* in the *Virtuoso Module Generator User Guide*.
- **(IC6.1.8 Only)** *Using the Cell Planner* in the *Virtuoso Analog Placer User Guide*.
- *The Process Rule Editor* in the *Virtuoso Unified Custom Constraints User Guide*.

Using a Custom Constraint Editor

In addition to being able to use the *Cell Planner* and *Module Generator* for constraint editing, you can plug in the constraint editor application interface to other third party editors.

The custom constraint editor can be registered by loading a SKILL file (which must have an `.il` extension) into the application or by storing the file in a `.cadence/dfII/ci/editors` directory.

As the following example shows, the *Module Generator* is *plugged-in* to the *Constraint Manager* user interface:

```
ciRegisterConstraintEditor(
        list(nil 'name "Module Generator"
        'description "Modgen - Layout Structure Editor"
        'constraintType "modgen"
        'constraintParams list( "type=module")
        'editorAvailableExpression "isCallable('mgCreateOrEdit)"
        'startEditorExpression "mgCreateOrEdit(geGetEditCellView()
                               constraint nil)"
        'iconName "nexGenEd"
        'addToToolbar t
        )
    )
```

Where:

- `name`

  Is the name of the new constraint editor.

- `description`

  Is a description of the new constraint editor.

- `constraintType`

Is the type of constraint that can be edited using the specified constraint editor. When you double-click the constraint of the specified type, the corresponding constraint editor opens for the required editing. If you select a constraint in the *Constraint Manager* that does not match this type, the constraint editor option will be grayed out for selection.

- `constraintParams`

  Is a list of constraint parameter names and values that require to be matched for the constraint editor to be able to edit a constraint in the *Constraint Manager*. If the parameter name or values do not match the settings made here, the constraint editor option will be grayed out.

- `editorAvailableExpressions`

  Is an expression that is evaluated to determine if a constraint editor is available in the application currently being run. For example, the *Module Generator* and *Cell Planner* are only available in the `layoutPlus`, and `virtuoso` workbenches. If this expression returns `nil`, the constraint editor will not be registered with the *Constraint Manager*, and a toolbar option will be created.

- `startEditorExpression`

  Is an expression evaluated in advance of launching the constraint editor. The constraint variable is available in the context of the expression, and is a reference to the constraint object selected in the *Constraint Manager* prior to the constraint editor option being selected. If no constraints were selected, the constraint will be `nil`.

- `iconName`

  Is the name of the `.png` file of the constraint editor icon that should be added to the *Constraint Manager* toolbar, for example `nexGenEd.png`.

  Note: This icon must exist in the icons search path. For more information, see Toolbar Definition File Search Path and Customization in the *Virtuoso Design Environment User Guide*.

- `addToToolbar`

  Determines whether or not a tool button be added to the *Constraint Manager* toolbar for the constraint editor. If `nil`, then the constraint editor will only be added to the list of constraint editor options in *Constraint Manager* table context menu.

  Note: When set as part of the `ciRegisterConstraintGenerator` SKILL function, a value setting of `addToToolbar nil` will ensure that no entry for that constraint is displayed in the *Constraint Generator* drop-down menu.

Important

For information on SKILL commands that can be used for *Constraints Manager* assistant customization, see Constraints SKILL API Commands in the *Virtuoso Unified Custom Constraints SKILL Reference*.

Configuring to Open Constraint Editor on Double-Click

By default, the defined constraint editor opens when you double-click a constraint of the type specified with the `constraintType` argument of the `ciRegisterConstraintEditor` SKILL function. However, to open a constraint editor when a template is double-clicked in the *Constraint Browser*, you need to configure that based on the settings you apply by using the `ciRegisterConstraintEditor` and `ciRegisterConstraintGenerator` SKILL functions. For detailed information, refer to the Defining the Interface for Editing a Template and its Parameters section in the *Virtuoso Unified Custom Constraints User Guide*.