Synopsys Design Constraints (SDC)
Open Source Parser

# SDC Parser User's Manual

# Contents

## 1. Introduction

Synopsys Design Constraints Format used as common format of Design Constraints representation for EDA tools.
SDC Parser supports reading SDC files by EDA tools. SDC Parser command-line shell also could be used as a checker of SDC files (see Figure 1).

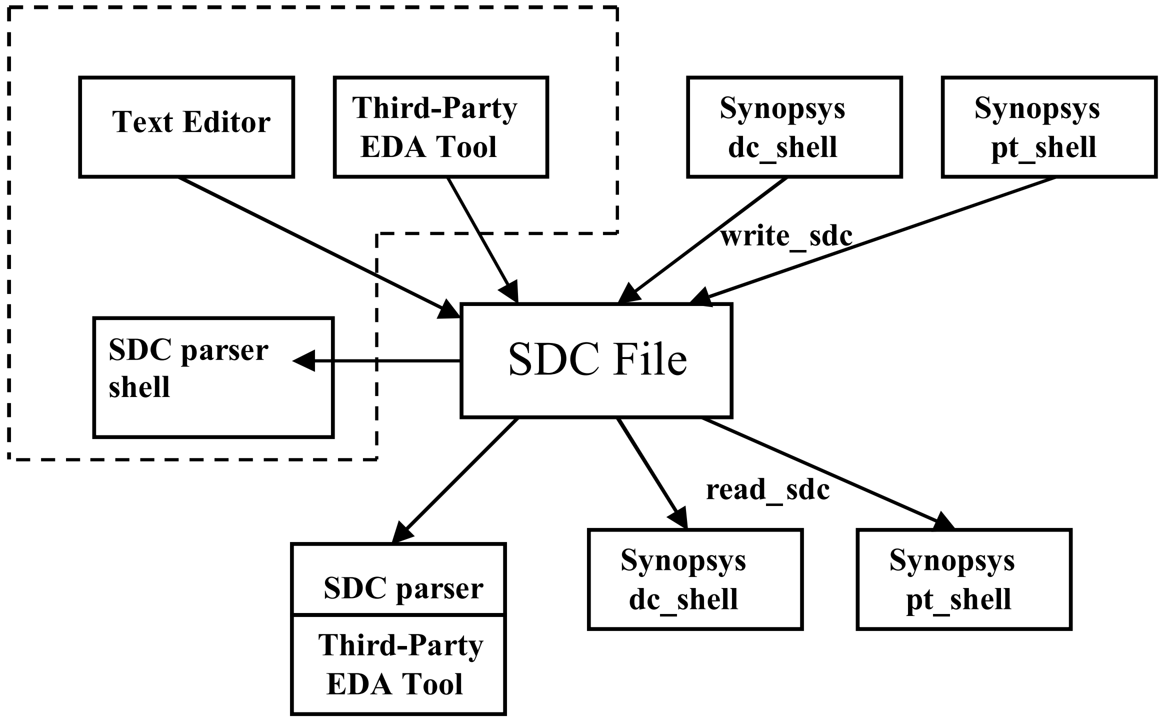

**Figure 1. SDC File Generation & Processing**

The main features of SDC parser are the following:

- It checks SDC file in accordance with SDC Spec and provides error messages
- It generates internal data structure convenient for processing by EDA tool
- It provides Tcl API and command-line interface for EDA tool
- It supports several SDC versions, which can be defined in SDC file or by EDA application
- It works at any platform supporting TCL 8.0 (or newer) Solaris, Linux, Windows NT/2000/98/95

## 2. Parser Architecture

### 2.1. SDC Parser Data Flow

The Data flow shown on Figure 2.

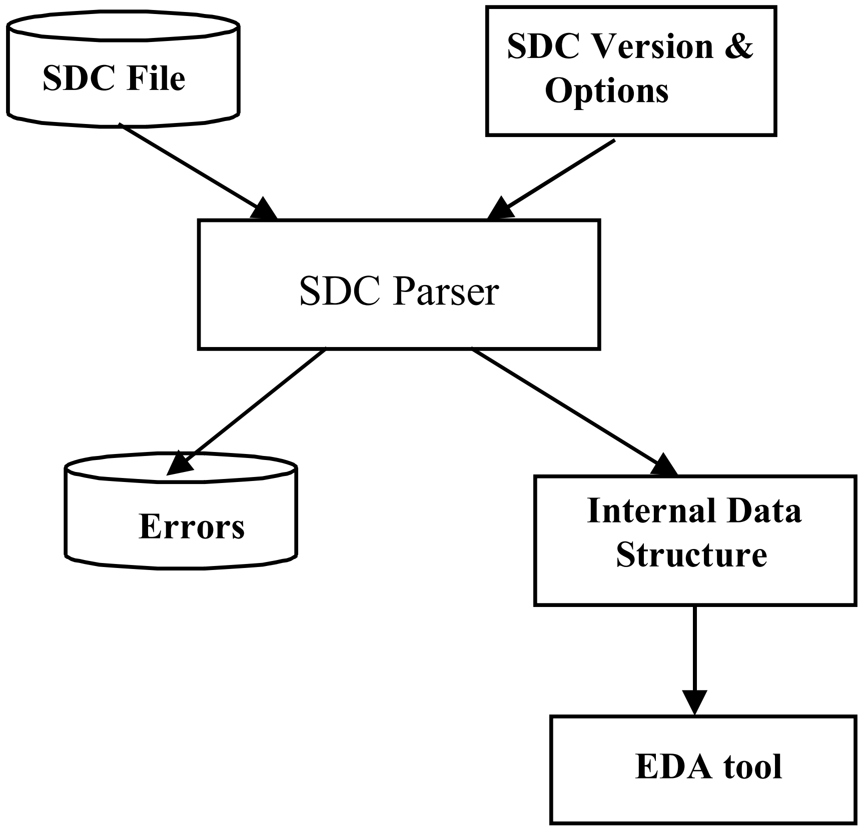

Figure 2. SDC Parser Data Flow

## 2.2. Program structure of SDC parser

Figure 3 shows the Program Structure.

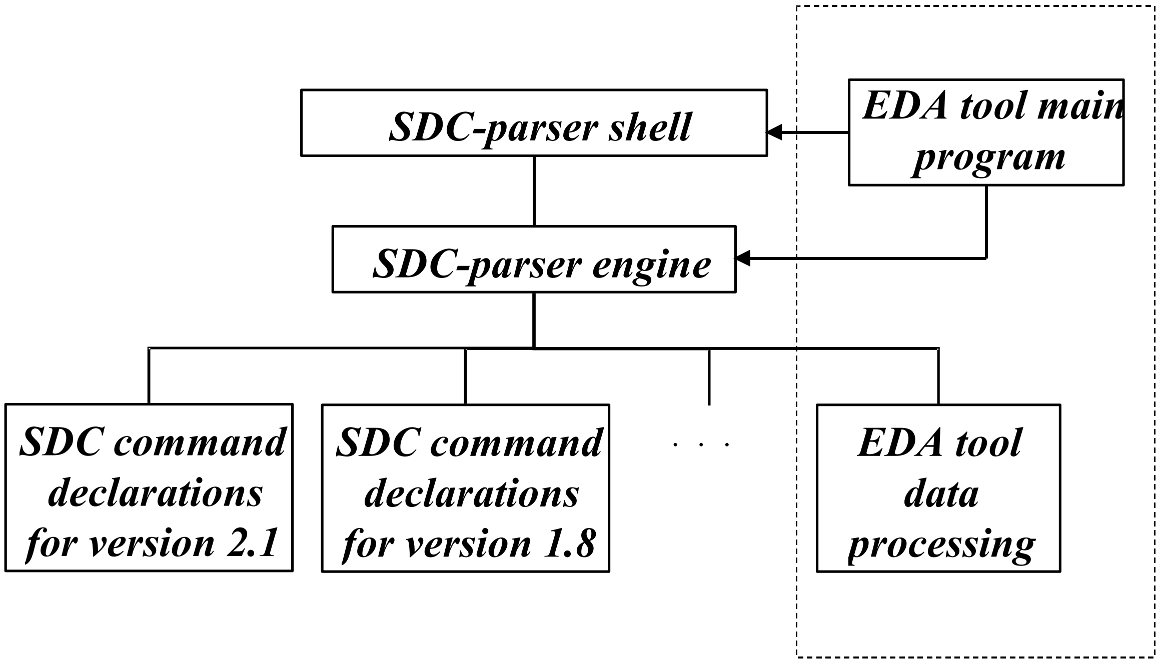

Figure 3. Program Structure of SDC Parser.

Descriptions of modules:

*SDC-parser shell*

Interface program for working with SDC-files from command prompt through SDC-parser engine API

*SDC-parser engine*

Contains API and general functions of SDC-parser

*SDC-parser declarations for version X.X*

Contains descriptions of SDC commands.
There is set of files for SDC versions supported by the parser.
Each file contains set of functions for supported by particular SDC version SDC commands in the format like this:

- name of the command
- list of parameters and properties of parameters for this command (name, type, mandatory or optional, restrictions)

*EDA tool process data procedure*

This procedure extracts data from data structure built by the parser and processes them in accordance with project specification.

## 2.3. Algorithm of SDC Parser.

General algorithm of SDC Parser presented in Figure 4.

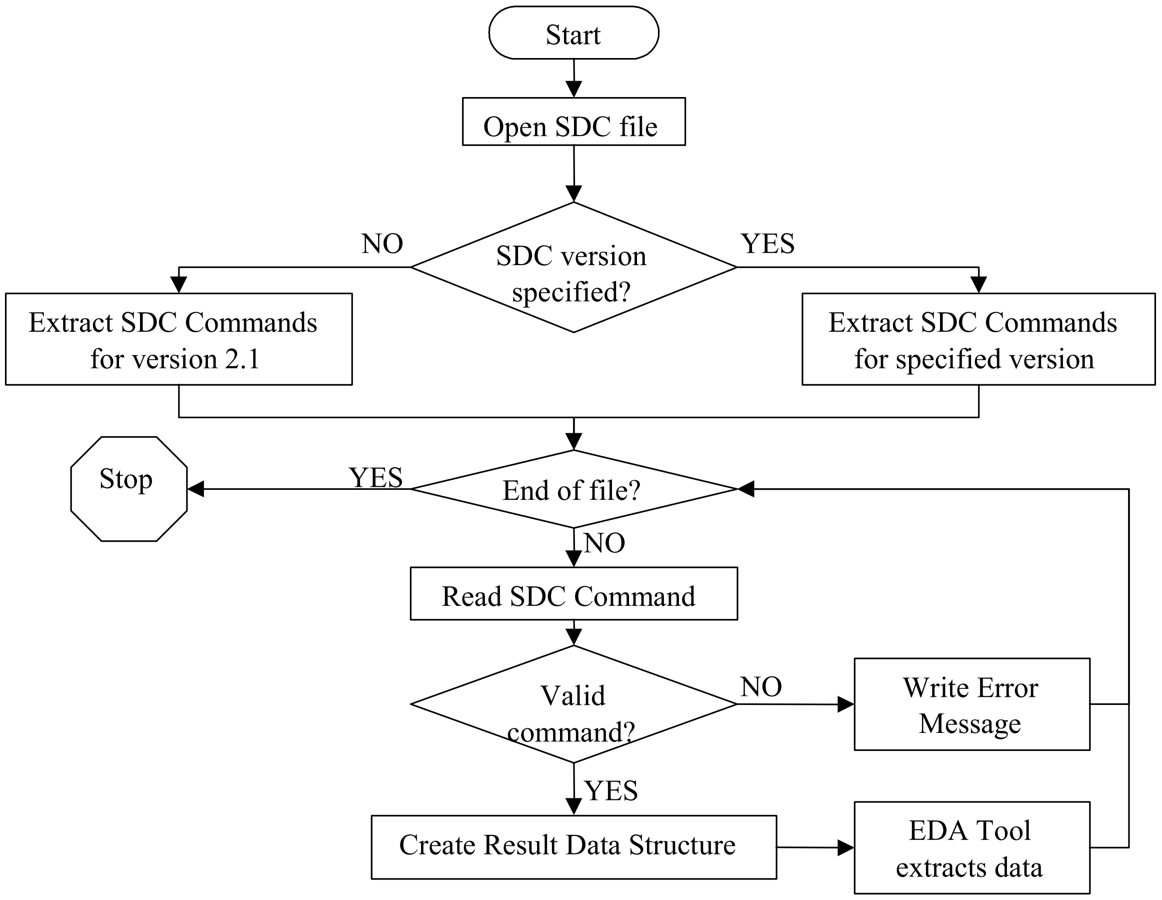

Figure 4. General algorithm of SDC Parser engine.

# 3. Parser Interface

EDA tool could use Parser Engine API or command-line interface.
Simple example of EDA tool main program with call-back procedure presented in Appendix B.

## 3.1 API Functions.

There is below list of API functions with their descriptions:

### sdc::parse_file

The **sdc::parse_file** function runs sdc-parser for reading SDC file. The syntax is:

> **sdc::parse_file {SDC_file_name}**

***SDC_file_name*** – name of SDC file (which contain SDC commands).

### sdc::parse_command

The **sdc::parse_command** function  runs sdc-parser for one SDC command and returns result of executing EDA callback function. The syntax is:

> **sdc::parse_command {version   command_name   args}**

***version*** – version number of SDC (Ex.: 2.0)
***command_name*** – name of the command (Ex.: create_clock)
***args*** – list of commands parameters

### sdc::declare

The **sdc::declare** function declare one SDC command (each SDC command should be declared for particular SDC version). The syntax is:

> **sdc::declare {command_name   arguments   [condition]}**

***command_name*** – name of the command (Ex.: create_clock)
***arguments*** - list of descriptions of commands parameters
***condition*** – condition describing correct set of SDC-command's arguments

Example of command declaration:

```
declare  create_clock {
  {-period          Float           {$par >= 0}}
  {-name            String          }
  {-waveform        List            }
  {port_pin_list    List            {type_Float}}
 } {param(-period)}
```

## sdc::register_callback

The **sdc::register_callback** function set a callback for EDA tool. The syntax is:

**sdc::register_callback {proc_name}**

***proc_name*** – name of "callback" procedure (if proc_name is empty callback will be disabled)

Syntax of callback procedure:

***proc_name {command_name   array_name}***
*EDA-tool process data and returns flag "continue parsing"*
***command_name*** *– name of SDC command*
***array_name*** *– name of array with data after parsing*

## sdc::set_version

The **sdc::set_version** function change a current SDC version number. The syntax is:

**sdc::set_version {sdc_version}**

***sdc_version*** – new version number

## sdc::out_message

The **sdc::out_message** function puts message to the console / log file depending on settings. The syntax is:

**sdc::out_message {message}**

***message*** – text of the message

### sdc::set_log_file

The **sdc::set_log_file** function sets name of log-file. The syntax is:

**sdc::set_log_file {filename}**

***filename*** – name of log-file

### sdc::set_debug_level

The **sdc::set_debug_level** function sets debug level. The syntax is:

**sdc::set_debug_level {level}**

***level*** – level of debug (0-2)

## 3.2 Command Line Interface.

To call SDC-parser shell use the following command line:

sdcparser.tcl [**options**] [**filename** [...]]

where

```
<filename>              - source file name;
-f  <filename>          - read command line arguments from file;
-d<debug_level>         - set debug level
        (d0 – no error messages, quiet mode,
         d1 – brief  error messages,
         d2 – extended error messages);

-l <log_file_name>      - set log file name;
-v <sdc_version>        - set default sdc version;
-eda <eda_source_file>  - file name which contains tcl source of eda tool procedure.
```

## 4. Data Structure After Parsing

After parsing, if callback was registered, callback procedure would be called with two parameters:

- SDC command name, which was parsed
- Name of associative array which contains parsed parameters with values (data structure)

For example, after parsing of command

**create_clock [ get_ports CLK2 ] -period 12.2 -name CLK2**

the data structure will look like

***data(-period)*** = 12.2
***data(-name)*** = CLK2
***data(port_pin_list)*** = result of function **get_ports**

See SDC Syntax description (Appendix A) to figure out the name of "non-dash" parameters.

## 5. SDC Versions Support

### 5.1. Declarations of SDC commands

SDC Commands are described in separate files for particular SDC version.

List of all supported versions presented in variable ***validsdcversions*** in file "*sdcparsercore.tcl*"
In format **[ list 2.1 2.0 1.9 1.8… ]**

File name for particular version has the following format: "*sdc<new_version>.tcl*" (Ex.: *sdc2.1.tcl*)

File structure is the sequence of SDC command declarations.

Format of command declarations is the following:

[sdc::]**declare {command_name arguments [condition]}**

Arguments have the following format:

```
{
{key_name    type  [additional_parameters]}
{key_name    type  [additional_parameters]}
…
{key_name    type  [additional_parameters]}
}
```

where

***key_name*** – name of key (if first char is "-", example: "-delay") or internal name of positional parameter.

***type*** – type of parameter. Valid values are:

| | |
|---|---|
| ***String*** | - value of this parameter must be a string type |
| ***Int*** | - value of this parameter must be an integer type, ***additional_parameters*** must be a expression (if no restrictions needed, put **"1"** to this parameter) |
| ***Float*** | - value of this parameter must be a float type, ***additional_parameters*** must be a expression (if no restrictions needed, put **"1"** to this parameter) |
| ***Flag*** | - no value for this parameter |
| ***Enum*** | - value of this parameter must be an enumerated string type, ***additional_parameters*** must be a list of valid values of parameter |
| ***List*** | - value of this parameter must be a list type, ***additional_parameters*** can be a list which contains additional flags and condition of list length |
| ***Unknown*** | - no checking of type will processed |

Examples:

```
declare create_clock  {
  {-period          Float     {$par>=0}}
  {-name            String     }
  {-waveform        List      {type_Float {length($length>=2 && ($length % 2)==0)}}}
  {port_pin_list    List      }
 }  {param(-period)}

declare set_min_delay {
  {delay_value      Float     {1}}
  {-rise            Flag      }
  {-fall            Flag      }
  {-from            List      }
  {-to              List      }
  {-through         List      {dup}}
 } {param(delay_value)}

declare get_cells  {
  {-of_objects      List          }
  {patterns         List          }
  {-hierarchical    Flag          }
 }  {(param(patterns) && !param(-of_objects)) || (param(-of_objects) &&
param(patterns) && !param(-hierarchical))}
```

Enabled combinations of arguments here are:

***patterns*** and no ***-of_objects***;

***-of_objects*** and no ***patterns*** and no ***-hierarchical***.

## Appendix A

For latest version SDC Commands description download “Using Synopsys Design Constraints Format Application Note” from Synopsys TAP-in SDC download selection.

## Appendix B

## Example of EDA application

```
#!/bin/sh
#
# This is a simple example of EDA application
#
# It contains
#
# - EDA callback procedure "callback_simple_example"
# - main program (register callback and parse file)
#
# The callback procedure
#   - prints parameter values for SDC commands
#     - create_clock
#     - set_input_delay
#   - returns parameter values for SDC Object Access
Functions
#     (here without searching in Design Data Base)
#     - get_clocks
#     - get_ports
#
#\
exec tclsh "$0" "$@"

# include parser engine

source [file join [file dirname [info script]]
sdcparsercore.tcl]

# callback procedure

proc callback_simple_example {command parsing_result} {

    # put reference to data structure after parsing

    upvar $parsing_result res

    # Switch on command type

    switch -- $command {


        create_clock -
        set_input_delay  {
            puts "Command: $command"
            foreach arg [array names res] {
                puts "  Argument $arg = $res($arg)"
            }
            puts ""
            return ""
        }


        get_clocks -
        get_ports {
            return $res(patterns)
        }

        default {
        }
    }
}

# main program

sdc::register_callback callback_simple_example

sdc::parse_file [lindex $argv 0]
```