

IEEE Standard for Integrated Circuit (IC) Delay and Power Calculation System

Sponsor

**Design Automation Standards Committee
of the
IEEE Computer Society**

Approved 26 June 1999

IEEE-SA Standards Board

Abstract: Ways for integrated circuit designers to analyze chip timing and power consistently across a broad set of electric design automation (EDA) applications are covered in this standard. Methods by which integrated circuit vendors can express timing and power information once per given technology are also covered. In addition, this standard covers means by which EDA vendors can meet their application performance and capacity needs.

Keywords: chip delay, electronic design automation (EDA), integrated circuit design, power calculation

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Copyright © 2000 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 12 May 2000. Printed in the United States of America.

Print: ISBN 0-7381-1771-4 SH94769
PDF: ISBN 0-7381-1772-2 SS94769

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

IEEE is the sole entity that may authorize the use of certification marks, trademarks, or other designations to indicate compliance with the materials set forth herein.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; (978) 750-8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Introduction

[This introduction is not part of IEEE Std 1481-1999, IEEE Standard for Integrated Circuit (IC) Delay and Power Calculation System.]

The objective of the delay and power calculation system (DPCS) is to make it possible for integrated circuit designers to *consistently* calculate chip delay and power across electronic design automation (EDA) applications, and for integrated circuit vendors to express delay and power information only *once* per technology while enabling sufficient EDA application accuracy.

This is accomplished by a coordinated set of standards that support a standard method to describe timing and power characteristics of integrated circuit design units (cells and higher level design elements); a standard method for EDA applications to calculate chip design instance specific delay, slew, and power for logic and interconnects; and standard file formats to exchange chip parasitic and cluster information.

Background

A major integrated circuit design problem is to ensure the chip meets the designer's timing and power requirements. At this time in the EDA industry, there are numerous methodologies—many incompatible or inconsistent with each other—for representing timing and power information. The DPCS responds to the urgency to improve chip design methods.

The DPCS includes all aspects and uses of delay and power calculation (including synthesis, floorplanning, and simulation) during both pre- and post-layout phases of chip design. A fundamental requirement placed on the DPCS is that it work equally well with Verilog HDL and VHDL. In this standard, the scope of the DPCS is limited to integrated circuit designs.

The DPCS gains its leverage out of an effort by Silicon Integration Initiative (Si2) to standardize contributed technology from IBM in the form of a delay calculation language (DCL) and its companion procedural interface (PI), along with an effort by Open Verilog International (OVI) to standardize file formats to exchange chip parasitic and cluster information in the form of a Physical Design Exchange Format (PDEF) and a Standard Parasitic Exchange Format (SPEF). SPEF is based on the Standard Parasitic Format (SPF) technology supplied by Cadence Design Systems. PDEF is based on technology provided by Synopsys.

Since each of these specifications is based on existing technology that has proven itself in industry, the time to introduce this standard into industry is greatly reduced over what would have been required for a newly created standard. The industry owes a great deal of gratitude to these companies for contributing these technologies.

This standard is targeted toward modeling integrated circuit cell delays and power at a level of accuracy sufficient for sign-off and a level of power sufficient to support design synthesis tools.

Compliance

Technology libraries and EDA tools shall be considered compliant with this standard if they correctly implement the DCL, PIs, and/or back-annotation file formats described in this document.

Vendors may implement what this standard defines as they see fit, provided the implemented functionality matches the definitions.

Organization

The Delay and Power Calculation Working Group is composed of four subgroups: Architecture, Language, Parasitics and Clustering, and Power. The members of each of these subgroups are members of the Delay and Power Calculation Working Group. The Delay and Power Calculation Working Group has a vice-chair liaison with Japan to coordinate development and review of the specification there. The IEC TC93 WG 2 also participates in the review and development of this specification.

In addition to numerous individual contributors, both OVI, VHDL International (VI), and Si2 have played significant roles in the development of this standard and funding the technical documentation.

Membership

The Delay and Power Calculation Working Group is responsible for issue analysis and resolution along with the wording of the proposed draft. The working group developed a power calculation method and specified a means for multiple delay and power calculation modules (DPCMs) to be used concurrently in one design so multiple implementations of the standard can work simultaneously. It would not have been possible to complete this effort without the dedication and hard work of this group.

About 100 representatives from more than 40 companies monitored and guided the development of the DPCS specification through their review of technical data supplied via electronic media.

Participants

At the time this standard was completed, the Delay and Power Calculation Working Group had the following membership:

Dennis Brophy, *Chair*
Jay Abraham, *Vice Chair*
Kumiko Ishiko, *Vice Chair—Japan*

Toshio Aizawa
Mike Andrews
Dave Barton
Karen Bartleson
Harry J. Beatty III
Rajesh R. Berigei
Ram Bhamidipaty
Tharini Chakraborty
Shir-Shen Chang
Ellis Cohen
Donald Cottrell
Graham Davies
Nitin Deo
Ashesh Doshi
Peter Elmendorf

James Engel
Laura Funke
Vassilios Gerousis
Kevin Grotjohn
Runip Gupisetty
Stuart Hamilton
Eric Howard
Tim Hsia
Anling Hsu
Tim Jennings
Tim Jordan
Ajay A. Kadakia
Adel Khouja
Archie Lachner

Dave Mallis
Steve Mecham
Harold E. Reindel
Karla Reynolds
Ann Marie Rincon
Kostas Siomalas
Sridhar Subramaniam
Andres Teene
Jesse Tessier
Yatin K. Trivedi
Hirendu Vaishnav
Q. Yan
Tak Young
Gaofeng Wang
Carl Wong

The following persons were on the balloting committee:

Mostapha Aboulhamid	Brian S. Griffin	Karla Reynolds
Jay Abraham	Andrew Guyler	Steffen Rochel
Takashi Aikyo	Gregory A. Haynes	Fred U. Rosenberger
Stephen A. Bailey	John Hillawi	Toshiyuki Saito
Pete Bakowski	John Hines	Karem A. Sakallah
David L. Barton	Yasuaki Inoue	K. Scott Sampson
Harry J. Beatty III	Kumiko Ishiko	Steven E. Schultz
Victor Berman	Takashi Kambe	Maitreya Sengupta
J. Bhasker	Osamu Karatsu	Ravi Shankar
William Billowitch	Jake Karrfalt	Charles F. Shelor
Dennis B. Brophy	Masamichi Kwarabayashi	Djahida Smati
John A. Busco	Satoshi Kojima	Mircea Stan
Theodor Calin	Serge Maginot	Atsushi Takahara
E. Camparini	Victor Martin	Andres Teene
Raul Camposano	R. Stephen Mecham	Jose A. Torres
Zoltan J. Cendes	Paul J. Menchini	Masahiko Toyonaga
Shir-Shen Chang	Jean P. Mermet	Yatin K. Trivedi
Luc Claesen	Laura Funke Miller	Srinivasa Vemuru
Ellis Cohen	Gabe Moretti	Ronald Waxman
Donald Cottrell	David S Morris	Cary Wei
Ted Craven	Akira Nagao	John M. Williams
Clifford E. Cummings	Hiroshi Nakamura	Carl K. Wong
Douglas D. Dunlop	Ryosuke Okuda	Lun Ye
Wolfgang Eisenmann	Yoichi Onishi	Hitoshi Yoshizawa
Peter Elmendorf	Timm Ostermann	Tak Young
James Engel	David M Parry	Tetsuo Yutani
Joerg-Oliver Fischer-Binder	William R Paulsen	John W. Zack
Suresh Gopalakrishnan	Alexander Perry	Alex N. Zamfirescu
Kenji Goto	Harold E. Reindel	Kristian M. Zoerhoff

When the IEEE-SA Standards Board approved this standard on 26 June 1999, it had the following membership:

Richard J. Holleman, *Chair*
Donald N. Heirman, *Vice Chair*
Judith Gorman, *Secretary*

Satish K. Aggarwal	James H. Gurney	Louis-François Pau
Dennis Bodson	Lowell G. Johnson	Ronald C. Petersen
Mark D. Bowman	Robert J. Kennelly	Gerald H. Peterson
James T. Carlo	E. G. "Al" Kiener	John B. Posey
Gary R. Engmann	Joseph L. Koepfinger*	Gary S. Robinson
Harold E. Epstein	L. Bruce McClung	Akio Tojo
Jay Forster*	Daleep C. Mohla	Hans E. Weinrich
Ruben D. Garzon	Robert F. Munzner	Donald W. Zipse

*Member Emeritus

Also included is the following nonvoting IEEE-SA Standards Board liaison:

Robert E. Hebner

Greg Kohn
IEEE Standards Project Editor

Windows NT[®] is a registered trademark of Microsoft Corporation.

Contents

1.	Overview.....	1
1.1	Scope.....	1
1.2	Purpose.....	2
1.3	Contents of this standard.....	2
2.	References.....	3
3.	Definitions.....	3
4.	Acronyms and abbreviations.....	11
5.	Delay and power calculation system architecture.....	12
5.1	Overview.....	12
5.2	Procedural interface.....	14
5.3	DPCM-application relationships.....	15
5.4	Interoperability.....	16
6.	Delay calculation language (DCL).....	16
6.1	Character set.....	17
6.2	Lexical elements.....	17
6.3	Name spaces of identifiers.....	28
6.4	Storage durations of objects.....	28
6.5	Scope of identifiers.....	28
6.6	Linkages of identifiers.....	29
6.7	DCL data types.....	29
6.8	Type conversions.....	32
6.9	Operators.....	33
6.10	Expressions.....	38
6.11	Computation order.....	44
6.12	DCL statements.....	46
6.13	Tables.....	57
6.14	Library control statements.....	69
6.15	Modeling.....	74
6.16	Embedded C code.....	91
6.17	Definition of a subrule.....	92
7.	Power modeling and calculation.....	93
7.1	Power overview.....	94
7.2	Caching state information.....	95
7.3	Caching load and slew information.....	95
7.4	Simultaneous switching events.....	97
7.5	Partial swing events.....	97
7.6	Power calculation.....	97
7.7	Accumulation of power consumption by the design.....	99
7.8	Group pin list syntax and semantics.....	99
7.9	Group condition list syntax and semantics.....	100
7.10	Sensitivity list syntax and semantics.....	101
7.11	Group condition language.....	102

8.	Procedural interface (PI).....	107
8.1	Overview.....	107
8.2	Control and data flow.....	108
8.3	Architectural requirements.....	108
8.4	Data ownership technique.....	108
8.5	Application/DPCM interaction	110
8.6	Re-entry requirements.....	112
8.7	Application responsibilities when using a DPCM	112
8.8	Application use of the DPCM.....	114
8.9	DPCM library organization.....	117
8.10	DPCM error handling	117
8.11	C-level language for EXPOSE and EXTERNAL functions.....	118
8.12	PIN and BLOCK data structure requirements	121
8.13	DCM_STD_STRUCT standard structure	121
8.14	DCMTransmittedInfo structure	125
8.15	Environment or user variables	125
8.16	PI functions summary	125
8.17	PI function descriptions	135
8.18	Standard structure (dcmstd_stru.h) file.....	230
8.19	Standard macros (dcmstd_macros.h) file	247
8.20	Standard interface structures (dcmintf.h) file	250
8.21	Standard loading (dcmload.h) file.....	255
8.22	Standard debug (dcmdebug.h) file	256
8.23	Standard array (dcmgarray.h) file	272
8.24	DCM user array defines (dcmuarray.h) file.....	276
8.25	Standard platform-dependency (dcmpltfm.h) file.....	279
8.26	Standard state variables (dcmstate.h) file	285
8.27	Standard table descriptor (dcmutab.h)	287
9.	Parasitics	288
9.1	Introduction.....	289
9.2	Targeted applications for SPEF	289
9.3	SPEF specification	289
9.4	Examples.....	316
10.	Physical design exchange	333
10.1	Introduction.....	334
10.2	PDEF specification	335
10.3	Examples.....	373
	Annex A (normative) Implementation requirements.....	381
	Annex B (informative) Calculation of total load capacitance in the DPCS	383
	Annex C (informative) Hold control	386
	Annex D (informative) Bibliography	391

IEEE Standard for Integrated Circuit (IC) Delay and Power Calculation System

1. Overview

The delay and power calculation system (DPCS) is a coordinated set of standards that support a standard method to describe timing and power characteristics of integrated circuit (IC) design units (cells and higher level design elements); a standard method for electronic design automation (EDA) applications to calculate chip design instance specific delay, slew, and power for logic and interconnects; and standard file formats to exchange chip parasitic and cluster information. The four distinct standard specifications covered in this document include

- A description language for timing and power modeling, delay calculation language (DCL).
- A software procedural interface (PI) for communications between EDA applications and compiled libraries of DCL descriptions.
- A standard file exchange format for parasitic information about the chip design: Standard Parasitic Exchange Format (SPEF).
- A standard file exchange format for floorplan cluster information relative to the chip design: Physical Design Exchange Format (PDEF).

1.1 Scope

The scope of the DPCS standard is to make it possible for integrated circuit designers to analyze chip timing and power consistently across a broad set of EDA applications, for integrated circuit vendors to express timing and power information once (for a given technology), and for EDA vendors to meet their application performance and capacity needs. The intended use for this standard is IC timing and power. This standard may be applied to both unit logic cells supplied by the IC vendor and logical macros defined by the IC designer. Although this standard is written toward the integrated circuit supplier and EDA developer, its application applies equally well to representation of timing and power for designer-defined macros (or hierarchical design elements).

These specifications make it *possible* to achieve consistent timing and power results, but do not *guarantee* it. They provide for a single executable software program that computes delay and power based on IC vendor-supplied algorithms (or designer-supplied algorithms for macros), but does not guarantee EDA applications will correctly communicate the design-specific information required for these algorithms. By specifying

standard exchange formats for parasitic data and floorplanning information, this standard provides a marked improvement over design environments with no such standards. However, it is the responsibility of the EDA application to correctly correlate the information between these standard exchange files and the actual design. This standard also does not detail how the information contained within the standard exchange files shall be obtained.

1.2 Purpose

As feature sizes for chips shrink below 0.5 μm , interconnect delay effects outweigh those of the logic cells. This means placement of cells and wire routing of the interconnects become as important a factor as the type of cell drivers and receivers on the interconnect. As a result, EDA logic design applications (such as synthesis) now need to interact closely with physical design applications (such as floorplanning and layout). Applications that before could consider only simple delay and power models now need to deal with complex delay and power equations. The designer now needs EDA applications that can be directed to specific timing constraints, and provide consistent and accurate characterization of a chip's timing and power before it is manufactured. Plus, due to the complexities of the delay and power equations, the integrated circuit vendor needs to have control of application calculations and not be restricted by unique characteristics of the broad set of applications demanded by the customers (or designers).

Over the past few years it has become increasingly apparent that modern very large scale integration (VLSI) design is no longer bounded only by timing and area constraints. Power has become significantly more important. In an era of hand-held devices, ranging from mobile computing to wireless communication systems, managing and controlling power takes on an important role.

Several benefits can be attained from low power designs in addition to extended battery life. Low-power devices often run at lower junction temperature, which leads to higher reliability and lower-cost cooling systems. There are also several challenges for calculation and modeling of power (and delay) in deep-submicron (less than 0.25 μm) designs. EDA tools can now accurately calculate and model power by using this DPCS standard.

1.3 Contents of this standard

The organization of the remainder of this standard is

- Clause 2 (References) provides references to other applicable standards that are assumed or required for DPCS.
- Clause 3 (Definitions) defines terms used throughout the different specifications contained in this standard.
- Clause 4 (Acronyms and abbreviations) defines the acronyms used in this standard.
- Clause 5 (Delay and power calculation system architecture) gives an overview of how all of the elements of this standard fit together to achieve the DPCS goals.
- Clause 6 [(Delay calculation language (DCL))] provides the specification for the language used to model timing and power.
- Clause 7 (Power modeling and calculation) provides the specification for modeling power and the interaction between an application and a delay and power calculation module (DPCM) for power calculations.
- Clause 8 [Procedural interface (PI)] provides the specification of the functions used to communicate between EDA applications and the DCL-based libraries, as well as their software interface.
- Clause 9 (Parasitics) provides the specification for the exchange of parasitics that are required to calculate accurate delays.

- Clause 10 (Physical design exchange) provides the specification for the exchange of floorplan clustering information that is used to provide more accurate timing calculations based on approximate layout.
- Annexes. Following Clause 10 are a series of normative and informative annexes.

2. References

This standard shall be used in conjunction with the following publication. When the following standard is superseded by an approved revision, the revision shall apply.

ISO/IEC 9899:1990, Programming Languages—C.¹

3. Definitions

For the purposes of this standard, the following terms and definitions apply. The IEEE Standard Dictionary of Electrical and Electronics Terms [B5] should be consulted for terms not defined in this standard.

3.1 application, electric design automation (EDA) application: Any software program that interacts with the delay and power calculation module (DPCM) through the procedural interface (PI) to compute instance-specific timing values. Examples include batch delay calculators, synthesis tools, floorplanners, static timing analyzers, etc. *See also:* **delay and power calculation module; procedural interface.**

3.2 arc: *See:* **timing arc.**

3.3 argument: The value or the address of a data item passed to a function or procedure by the caller.

3.4 back-annotation: The annotation of information from further downstream steps (toward fabrication) in the design process. *See also:* **back-annotation file.**

3.5 back-annotation file: A file containing information to be read by a tool for the purpose of back-annotation, for example Physical Design Exchange Format (PDEF) and Standard Parasitic Exchange Format (SPEF) files. *See also:* **back-annotation; timing annotation.**

3.6 bias: The time difference between the data arrival time and a specified signal edge (e.g., of a clock). Also, the BIAS clause used in a CHECK statement.

3.7 bidirectional: A pin or port that can place logic signals onto an interconnect and receive logic signals from it (i.e., act both as a driver and a receiver).

3.8 bus: In PDEF, a physical collection of nets and/or pnets, or of pins and/or nodes. If the items collected in the PDEF bus are logical, the PDEF bus may or may not correspond to a logical bus described in the netlist.

3.9 C-effective: A capacitance value, often computed as an approximation to a resistor/inductor/capacitor (RLC) network or a π -model, that characterizes the admittance of an interconnect structure at a particular driver. The reduction of real parasitics and pin capacitances to a C-effective allows the calculation of delay

¹ISO publications are available from the ISO Central Secretariat, Case Postale 56, 1 rue de Varembe, CH-1211, Genève 20, Switzerland/Suisse (<http://www.iso.ch/>). IEC publications are available from the Sales Department of the International Electrotechnical Commission, Case Postale 131, 3, rue de Varembe, CH-1211, Genève 20, Switzerland/Suisse (<http://www.iec.ch/>). ISO/IEC publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

and slew values from cell characterization data which assumes a pure capacitive output load. *Syn:* **effective capacitance**.

3.10 cell: A primitive in an integrated circuit library. For the purposes of IEEE Std 1481-1999, primitive means the timing properties of the cell are directly described in the DPCM without reference back to the application for the internal structure of the cell. This primitiveness typically is a result of the characterization of that cell by the semiconductor vendor, but it may instead be a result of the construction of a timing model for a subcircuit by the application, and its loading into the DPCM at run-time. For PDEF, cell refers to a logical instance, and a library primitive is called a gate.

The term cell can arise in the context of the abstraction of a type of cell available in the library or in the concrete selection and placement of a cell in the final design. If the context is not clear, the terms cell type and cell instance (or just instance) shall be used. *See also:* **cell type; instance; cell instance**.

3.11 cell type: Name used to identify a particular cell in the library.

3.12 cluster: A grouping of cell instances and/or clusters that are constrained to each other due to physical location or some other shared characteristic(s). It is not valid to have a cell instance explicitly made a member of more than one cluster. *Syn:* **partition, region**.

3.13 column: In a Physical Design Exchange Format (PDEF) datapath cluster, a cluster of cell, spare_cell, and/or cluster instances placed or constrained to be placed in the vertical (Y-axis) direction. *See also:* **datapath; row**.

3.14 constraint: A timing property of a design that is supplied as a goal or objective to an electronic design automation (EDA) tool, such as logic synthesis, floorplanning, or layout. The tool shall not start out with a fixed design implementation; it shall build or modify the design to meet the constraint. *See also:* **timing check**.

3.15 datapath: A type of Physical Design Exchange Format (PDEF) cluster that contains rows and/or columns of cluster, cell, skip, and/or spare_cell instances. A PDEF datapath typically corresponds to structured logic. *See also:* **column; row**.

3.16 delay: The time taken for a digital signal to propagate between two points.

3.17 delay and power calculation module (DPCM): A software component compliant with the delay and power calculation system (DPCS) supplied by a semiconductor vendor that is responsible for computing instance-specific timing data under control of an EDA application. The DPCM is loaded into memory at run-time and linked to the application via the procedural interface (PI). A DPCM typically is created from DCL subrules compiled by the DCL compiler and linked together with run-time support modules.

3.18 delay and power calculation system (DPCS): The complete system detailed in IEEE Std 1481-1999: the DCL language, the procedural interface (PI) for delay and power calculations, and text formats for physical design and parasitic information.

3.19 delay arc: *See:* **timing arc**.

3.20 delay calculation language (DCL): The programming language used to calculate instance-specific timing data. DCL contains high-level constructs that can refer to the aspects of the design topology that influence timing and also express the sequence of calculations necessary to compute the desired delay and timing check limit values.

3.21 delay calculation language compiler: A software program, used in conjunction with a C compiler, that reduces DCL from ASCII text to computer executable format. *See also:* **delay and power calculation module.**

3.22 delay equation: Any mathematical expression describing cell delay or interconnect delay.

3.23 driver: A pin of a cell instance that, in the current context, is placing or can place a signal onto an interconnect structure.

3.24 early mode: The very first edge that propagates through a given cone of logic.

3.25 fanout: The pin count of a net (the number of pins connected to the net), minus one. This includes all input, output, and bidirectional pins on the net with the sole exception of one pin (assumed to be related to the particular timing arc currently of interest). Although less fundamental than pin count, fanout is frequently used in the definition of wireload models.

3.26 forward annotation: The annotation of information from further upstream (earlier in the design flow) in the design process. *See also:* **forward annotation file.**

3.27 forward annotation file: A file containing information to be read by a tool for the purpose of forward annotation, for example a Standard Delay Format (SDF) file containing PATHCONSTRAINTS. *See also:* **forward annotation.**

3.28 function, procedural interface (PI) function: One of the C functions that comprise the DPCS procedural interface.

3.29 gap: In Physical Design Exchange Format (PDEF), the spacing between rows and/or columns in a datapath.

3.30 gate: In Physical Design Exchange Format (PDEF), the physical abstraction of a library primitive.

3.31 hard macro: A cluster whose cell placements relative to each other are fixed. Often the interconnect routing between the cells is also fixed and a parasitics file describing the interconnect is available for the hard macro. The location of the hard macro in the floorplan may or may not be fixed.

3.32 hard region: A cluster that has defined physical boundaries in a floorplan. All cells contained in the cluster shall be placed within the boundaries of the cluster.

3.33 hierarchical instance: The concrete appearance of a design unit at some hierarchical level. Because higher-level design units may be instantiated multiple times, a single such appearance may give rise to multiple instances of the lower-level design units within it. Where instances are referred to as “occurrences,” hierarchical instances are referred to simply as instances.

3.34 hold time: *See:* **nochange timing check; setup/hold timing check.**

3.35 hold timing check: A timing check that establishes only the end of the stable interval for a setup/hold timing check. If no setup timing check is provided for the same arc, transitions, and state, the stable interval is assumed to begin at the reference signal transition and a negative value for the hold time is not meaningful. *See also:* **setup/hold timing check.**

3.36 input: A pin or port that shall only receive logic signals from a connected net or interconnect structure.

3.37 instance, cell instance: A particular, concrete appearance of a cell in the fully expanded (flattened, unfolded, elaborated) design description of an integrated circuit; also referred to elsewhere as an

“occurrence.” An instance is a “leaf” of the unfolded design hierarchy. In Physical Design Exchange Format (PDEF), this is a physical cluster or a logical cell. *See also:* **cell**, **cell type**, **cluster**, **hierarchical instance**.

3.38 interconnect: A collective term for structures (in an integrated circuit) that propagate a signal between the pins of cell instances with as little change as possible. These structures include metal and polysilicon segments, vias, fuses, antifuses, etc. Interconnect shall not include such structures if they occur as part of the fixed layout of a cell.

3.39 late mode: The very last edge that propagates through a given cone of logic.

3.40 layer: In Physical Design Exchange Format (PDEF), a particular level of interconnect on which a logical or physical pin is located.

3.41 library (integrated circuit): A collection of circuit functions, implemented in a particular integrated circuit technology, that an integrated circuit designer or electric design automation (EDA) synthesis application can select in order to implement a design. *See also:* **cell**.

3.42 library (software): A collection of object code units that may be linked, either statically or at run-time, with other libraries and/or object code modules to produce a software program.

3.43 library control statements: These statements control the logical organization and loading of subrules in a technology library. *See also:* **technology library**.

3.44 load-dependent delay: That part of a delay through a cell instance attributed to the admittance (load) presented to the arc sink pin and the internal impedance of the output.

3.45 mesh table: A multidimension table that defines every type of delay model in terms of discrete points. Each point represents a delay value in terms of several cell parameters or interconnect parameters. The delay calculation module is expected to interpolate between these points based on a mathematical expression defined by the technology file.

3.46 (to) model a cell: The creation of a specific elaboration of a model using `modelSearch`.

3.47 modeling procedures: The action of a circuit with respect to timing and power. These actions include creating segments and nodes, determining the propagation properties, and setting the delay and slew equations to use.

3.48 modeling statements: Delay calculation language (DCL) statements that map cell configurations to modeling procedures.

3.49 net, net instance: An abstraction expressing the idea of an electrical connection between various points in a design. In a hierarchical representation of the design, nets can occur at all levels and may connect to pins of lower hierarchical levels (including cell instances), ports of the current hierarchical level, and each other. In a flattened (unfolded and elaborated) design, electrically connected nets are collapsed and each net instance corresponds to an unique interconnect structure in the implementation.

3.50 nochange timing check: A timing check similar to a setup/hold timing check except the setup and hold times are referred to opposite transitions of the reference signal. The stable interval is extended to include the period between these transitions; that is, the time for which the reference signal stays in a specified state. This timing check is frequently applied to memory banks and latch banks to establish the stability of the address or to select inputs before, during, and after the write pulse.

3.51 node: A conceptual point (through which logic signals pass) that has been identified as an aid to modeling the timing properties of a cell but may not correspond to any physical structure. In Physical Design Exchange Format (PDEF), this is a physical pin that does not correspond to a logical structure.

3.52 nugget: A data structure used in the procedural interface (PI) for rapid switching between technologies.

3.53 output: A pin or port shall only place logic signals onto a connected net or interconnect structure.

3.54 parameter: A data item required for the calculation of some result.

3.55 parasitics: Electrical properties of a design (resistance, capacitance, and impedance) that arise due to the nature of the materials used to implement the design.

3.56 period timing check: A timing check that specifies the allowable time between successive periods of a signal.

3.57 periphery: The outer part of an integrated circuit where instances of cell types designed specifically to interface the internal circuitry to the “outside world” are placed. This part includes “pad” cells (which are input and output buffers) and power and ground pads; it may also include test circuitry, such as boundary scan cells.

3.58 π -model: A simplification of a general resistor/inductor/capacitor (RLC) network that represents the driving point admittance for an interconnect.

3.59 pin: A terminal point where an interconnect structure makes electrical contact with the fixed structures of a cell instance, or the conceptual point where a net connects to a lower level in the design hierarchy.

3.60 pin count: The number of cell instance pins that an interconnect structure visits, including all input, output, and bidirectional pins. Pin count is the number of “places” the interconnect goes to on the chip.

3.61 pnet: A physical net that has no correspondence to the logical function of the design, such as a route segment, which is reserved for future routes across a hard macro, or a power net not described in the design netlist.

3.62 pole: The complex frequency where a Laplace transform is infinite. Combined with residues, this is a convenient mathematical notation for the impedance or transfer function of a passive circuit, such as a resistor/inductor/capacitor (RLC) circuit, since poles above this frequency can be ignored in calculations without significant loss of accuracy.

3.63 port: A conceptual point at which a cell or a hierarchical design unit makes its interface available to higher levels in the design hierarchy.

3.64 primary input: The point where a logic signal arrives at the boundary of the design as currently known to an electric design automation (EDA) application. For a complete integrated circuit design, for example, this point is the metal pad of an input or bidirectional pad cell.

3.65 primary output: The point where a logic signal leaves the design as currently known to an electric design automation (EDA) application. For a complete integrated circuit design, for example, this point is the metal pad of an output or bidirectional pad cell.

3.66 procedural interface (PI): The set of C functions used by an application and a delay and power calculation module (DPCM) to exchange information and determine the timing calculation for a design.

3.67 pulse width timing check: A timing check that specifies the minimum time a signal shall remain in a specified state once it has transitioned to that state.

3.68 resistance times capacitance (RC), RC time constant: The product of some resistance and some capacitance (having the dimensions of time) or a time constant computed in some other way. *Syn:* **Elmore delay**.

3.69 receiver: A pin of a cell instance that is receiving or can receive a signal from an interconnect structure.

3.70 recovery/removal timing check: A timing check that establishes an interval with respect to a reference signal transition, during which an asynchronous control signal may not change from the active to inactive state. This timing check is frequently applied to flip-flops and latches to establish a stable interval for the set and reset inputs with respect to the active edge of the clock or the active-to-inactive transition of the gate.

Two limit values are necessary to define the stable interval. The recovery time is the time before the reference signal transition when the stable interval begins. The removal time is the time after the reference signal transition when the stable interval ends.

If the asynchronous control signal goes inactive during the stable interval, it is unknown whether the flip-flop or latch takes on the state of the data input, remains set, or is reset.

3.71 recovery time: *See:* **recovery/removal timing check**.

3.72 recovery timing check: A timing check that establishes only the beginning of the stable interval for a recovery/removal timing check. If no removal timing check is provided for the same arc, transitions, and state, the stable interval is assumed to end at the reference signal transition, and a negative value for the recovery time is not meaningful. *See also:* **recovery/removal timing check**.

3.73 region: A region pertains to a particular physical section or block of a floorplan. *See:* **cluster**.

3.74 removal time: *See:* **recovery/removal timing check**.

3.75 removal timing check: A timing check that establishes only the end of the stable interval for a recovery/removal timing check. If no recovery timing check is provided for the same arc, transitions, and state, the stable interval is assumed to begin at the reference signal transition, and a negative value for the removal time is not meaningful. *See also:* **recovery/removal timing check**.

3.76 residue: The value of $\lim_{s \rightarrow s_0} (s - s_0) \times F(s)$, where $F(s)$ has the complex pole s_0 . *See also:* **pole**.

3.77 return code: A value returned by a function indicating whether the function completed successfully. If the function did not complete successfully, it may return a nonzero return code; the exact value may indicate one of several possible severity conditions: informational, warning, error, severe, terminal error, etc.

3.78 route, global route: In Physical Design Exchange Format (PDEF), the physical description of interconnect routing between logical and physical pins of cell, spare_cell, and/or cluster instances.

3.79 row: In a Physical Design Exchange Format (PDEF) datapath cluster, a cluster of cell, spare_cell, and/or cluster instances placed or constrained to be placed in the vertical (Y -axis) direction. *See also:* **column**; **datapath**.

3.80 scalar: An integer constant.

3.81 segment: A portion of an interconnect structure treated as a unit for the purposes of extracting or estimating its electrical properties. *See also:* **parasitics**.

3.82 sequence point: A certain point in the execution sequence of a program where all side effects of previous evaluations are complete and no side effects of subsequent evaluations have occurred.

NOTE—Refer to section 5.1.2.3 of ISO/IEC 9899:1990.²

3.83 setup/hold timing check: (A) A timing check that establishes an interval with respect to a reference signal transition during which some other signal may not change value. This timing check is frequently applied to flip-flops and latches to establish a stable interval for the data input with respect to the active edge of the clock or the active-to-inactive transition of the gate.

Two limit values are necessary to define the stable interval. The setup time is the time before the reference signal transition when the stable interval begins and shall be negative if the stable interval begins after the reference signal transition. The hold time is the time after the reference signal transition when the stable interval ends, and shall be negative if the stable interval ends before the reference signal transition.

If the data signal changes during the stable interval, the reliability of the resulting state of the flip-flop or latch is unknown.

3.84 setup time: *See: nochange timing check; setup/hold timing check.*

3.85 setup timing check: A timing check that establishes only the beginning of the stable interval for a setup/hold timing check. If no hold timing check is provided for the same arc, transitions, and state, the stable interval is assumed to end at the reference signal transition and a negative value for the setup time is not meaningful. *See also: setup/hold timing check.*

3.86 shared port: An output or bidirectional port where some other output port of the cell derives its logic function. The output load at a shared port affects not only the delay to that port itself, but also the delay to any ports sharing it.

3.87 sink, sink pin: This is the end of a delay arc; that is, the destination of the logic signal. For arcs across cell instances, the sink is the driver pin; for arcs across interconnect, the sink is the receiver pin.

3.88 size metric: A value used to estimate properties of interconnect wholly contained in a region. The metric may be freely chosen (for example, square microns or gate sites), but it needs to be consistent between the cells and the wireload models. *See also: wireload model.*

3.89 skew timing check: A timing check that specifies the maximum time between two signal transitions. This timing check is frequently applied to dual-clock flip-flops to specify the maximum separation of the active edges of the two phases of the clock.

3.90 skip: In Physical Design Exchange Format (PDEF), skip is the spacing between the ordered cell, spare_cell, and/or cluster instances in rows and/or columns of a datapath.

3.91 slew: A measure of the shape of the waveform constituting a logic state transition. A slew value can have the dimensions of time, in which case it is a slew time, or the dimensions of voltage-per-time, in which case it is a slew rate. The delay and power calculation system (DPCS) allows either interpretation if used consistently.

3.92 slew-dependent delay: That part of an input-to-output delay that can be attributed to the signal at the input of the arc taking longer to make a transition than is considered ideal.

²Information on references can be found in Clause 2.

3.93 slew rate: A measure of how quickly a signal takes to make a transition; that is, a voltage-per-unit time. Slew rate is inversely related to slew time and is sometimes used incorrectly where slew time is intended.

3.94 slew time: A measure of how long a signal takes to make a transition; that is, the rise time or fall time. Slew time is inversely related to slew rate. The way a slew time value is abstracted from the continuous waveform at a cell pin varies with different cell characterization methods.

3.95 soft region: A cluster which does not have a specified physical location in a floorplan. It may have constraints on how closely the cells within the cluster are placed relative to each other. A soft region may be located within a hard region.

3.96 software implementation-defined behavior: Behavior, for a correct program construct and correct data, that depends on the software implementation and that each implementation shall document. The range of possible behaviors is delineated by IEEE Std 1481-1999.

3.97 software implementation limits: Restrictions imposed by an implementation.

3.98 source, source pin: The start of a delay arc; that is, the origin of the logic signal. For arcs across cell instances, the source is the receiver pin; for arcs across interconnect, the source is the driver pin.

3.99 spare_cell: A cell instance that is presently not part of the logical function of a design, and therefore is not included in the design's logical netlist. A spare_cell is typically reserved for future logic modifications to be implemented through changes in the interconnect layers of the chip.

3.100 standard structure: A particular C structure, defined in `std_stru.h`, that contains fields used to pass data over the procedural interface (PI) (thus avoiding large numbers of arguments). Most functions of the PI have a pointer to a *standard structure* as their first argument.

3.101 technology data: Data used to calculate the timing properties of a cell instance based on its context in the design. This term includes information that is not cell type specific or data specific for each cell type in the library. The kind of data used varies with the timing calculation methodology. General data and cell data may be contained in the same file or in separate files. Cell data also may be merged with the timing models of each cell, for example, when a tool performs its own timing calculation.

3.102 technology library: A technology library is a program written in delay calculation language (DCL) consisting of one or more subrules, each of which may contain references to other subrules (yet to be loaded). There is no hierarchical limit to the nesting of subrules within the scope of a technology library. Subrules can also be segmented into technology families, which alters the way they are made available to the application.

3.103 time-of-flight: The time delay between a signal leaving a driving pin or primary input port and reaching a receiving pin or primary output. Time-of-flight is generally dominated by the time taken to charge the distributed capacitance of the interconnect and the capacitance of the driven pins through the distributed impedance of the interconnect. The internal impedance of the driving port affects the load-dependent delay but *not* (directly) the time-of-flight.

3.104 timing annotation (file): The annotation of a design in one tool with timing data computed by another tool. If timing calculation is performed as an off-line process (separately from the application using the timing data), the process of reading the timing data into the tool is known as timing annotation. A timing annotation file stores the data written by the timing calculator and is later read by an application. Syn: **back-annotation**.

3.105 timing arc: A pair of ports, pins, or nodes possess some timing relationship, such as the propagation delay of a signal from one to the other or a timing check between them. Delay arcs may be between two distinct ports or nodes of a cell or over the interconnect from driver pins to receiver pins.

3.106 timing calculation, delay calculation: The process of calculating values for the delays and timing checks associated with the physical primitives (cells) of an integrated circuit design, or part of an integrated circuit design, and their interconnections.

3.107 timing check: A timing property of a circuit (frequently a cell) that describes a relationship in time between two input signal events. This relationship needs to be satisfied for the circuit to function correctly.

3.108 timing model: This represents the timing behavior of a cell for applications, such as simulation and timing analysis. For black-box timing behavior, it represents the definition of pin-to-pin delays between any pair of pins as well as internal nodes. In addition, for sequential cells it provides the definition of timing checks and constraints on any pair of pins and/or internal nodes.

3.109 transition: The change of a logic signal from one state to another (as in “...a transition at the input shall cause...”) or the pair of logic states between which a transition may occur (as in “...the delay for a low-to-high transition...”).

3.110 undefined behavior: Behavior for which the standard imposes no requirements (e.g., use of an erroneous program construct). Permissible undefined behavior ranges from:

- Ignoring a situation completely with unpredictable results.
- Behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message).
- Terminating a translation or execution (with the issuance of a diagnostic message).

NOTE—Many erroneous program constructs do not engender undefined behavior; they are required to be diagnosed.

3.111 unloaded delay: The conceptual delay value for a delay arc of a cell when the output pin is unloaded (unconnected) and the signal at the input pin conforms to some ideal waveform. *Syn:* **intrinsic delay**.

3.112 unspecified behavior: Behavior (for a correct program construct and correct data) that depends on the implementation. The implementation is not required to document which behavior occurs. Usually the range of possible behaviors is delineated by the standard.

3.113 via: In Physical Design Exchange Format (PDEF), a physical connection between two different levels of interconnect, or between a level of interconnect and a physical or logical pin.

3.114 wireload model: A statistical model for the estimation of interconnect properties as a function of the geometric measures available before the completion of layout and routing. Typical model properties include fanout, capacitance, length, and resistance. *See also:* **size metric**.

4. Acronyms and abbreviations

This clause lists the acronyms and abbreviations used in this standard.

API	application programming interface
ASIC	application specific integrated circuit
AWE	asymptotic waveform evaluation
CAE	computer-aided engineering [the term electronic design automation (EDA) is preferred]

CFI	CAD Framework Initiative
DCL	delay calculation language
DPCM	delay and power calculation module
DPCS	delay and power calculation system
D-SPF	Detailed Standard Parasitic Format
EDA	electronic design automation
EDIF	Electronic Design Interchange Format
HDL	hardware description language
OID	object identifier
OVI	Open Verilog International
PDEF	Physical Design Exchange Format
PI	procedural interface
PVT	process/voltage/temperature
RC	resistance times capacitance
RICE	rapid interconnect circuit evaluator
R-SPF	Reduced Standard Parasitic Format
SDF	Standard Delay Format
Si2	Silicon Integration Initiative
SPEF	Standard Parasitic Exchange Format
SPF	Standard Parasitic Format
SPICE	Simulation Program with Integrated Circuit Emphasis
VHDL	VHSIC Hardware Description Language
VHSIC	very-high-speed integrated circuit
VI	VHDL International
VITAL	VHDL Initiative Towards ASIC Libraries
VLSI	very-large-scale integration

5. Delay and power calculation system architecture

5.1 Overview

The goals of the DPCS architecture are to make it possible in a multivendor, multitool environment for integrated circuit designers to calculate timing and power *consistently*, for integrated circuit vendors to express timing and power information *once* for a given technology, while enabling sufficient EDA application accuracy.

To meet these goals, the DPCS shall have total control of delay and power calculation, support arbitrary expressions for delay and power values, and have very high performance. In addition, the DPCS architecture shall permit integrated circuit vendors to supply delay and power calculation to their customers independent of the release of design tools by EDA vendors.

Figure 1 shows a high-level representation of the DPCS architecture and relationships among several key components for calculating delay and power in an integrated circuit design environment.

The DPCM contains code (compiled from DCL source) that enables an application to compute power, timing delays, and timing constraints efficiently.

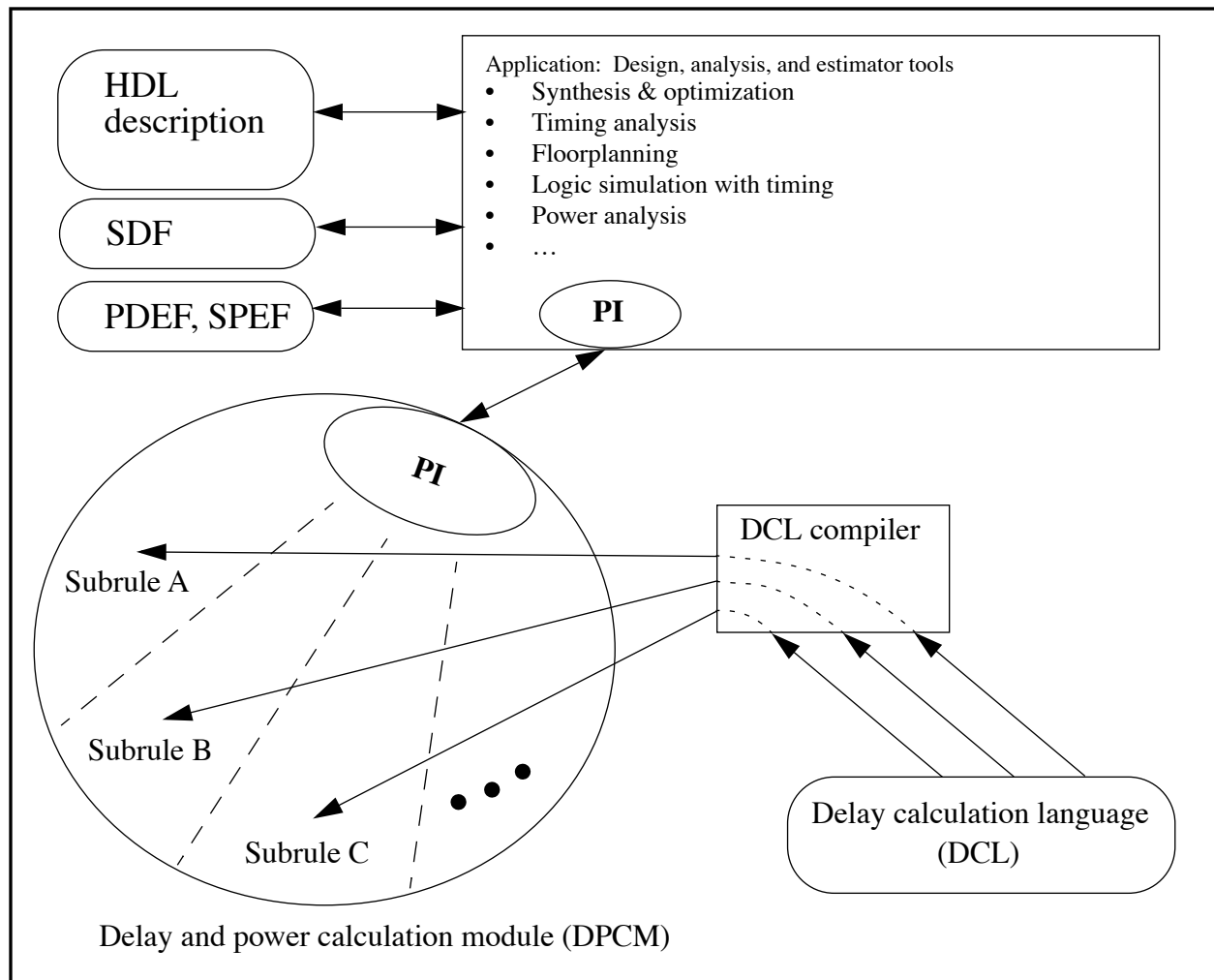


Figure 1 – High-level DPCS architecture

- The procedural interface (PI) is used by both an application and the DPCM to control delay and power calculations.
- Standard Delay Format (SDF) contains constructs for describing computed timing data (for back-annotation) and specifying timing constraints (for forward annotation). In integrated circuit design, it is common for a delay calculator application to calculate delays from post-layout data and write out the results in SDF.
- The Physical Design Exchange Format (PDEF) contains constructs to describe physical design information. It can be used as a back-annotation medium for passing physical design information from back-end applications (floorplanning and layout) to those on the front end; it can also be used to communicate physical constraints from front-end applications (synthesis, timing analysis, and partitioning) to those on the back end.
- The Standard Parasitic Exchange Format (SPEF) contains constructs to convey information between parasitic extractors and applications that utilize the extracted circuit information.

Figure 1 also illustrates several of the following key points:

- a) Applications shall be written (or modified) to use the DPCS-defined PI for all delay and power calculations.
- b) The application is fully responsible for mapping between design instances and delay models.
- c) The application is fully responsible for the state of the design. The DPCM knows everything about calculating delays and power, but nothing about a specific design. The DPCM uses the PI to acquire *all* design-specific information, including electrical modeling of parasitic effects and other data (often contained in SDF, PDEF, or SPEF files).
- d) The DPCM is composed of one or more *subrules*. Subrules are compiled C code constructed from an ASCII description following the rules of the DCL. The DPCM is dynamically linked with the application during execution.
- e) A subrule may implement one or more timing and power models.
- f) One or more timing models may be incrementally patched through the release (and use) of multiple subrules.

5.2 Procedural interface

A crucial part of the DPCS standard is the PI between an application and the DPCM. This PI enables the same DPCM to function with multiple applications. Details of the PI are described in Clause 8.

To enable an application to be independent of implementation-specific characteristics of a DPCM, code that implements the calls `dcmSetNewStorageManager`, `dcmSetMessageIntercept`, and/or `dcmBindRule` shall not be statically linked with the application. Figure 2 illustrates the required linkage structure. The code for these three functions shall reside in a shared object-code library named `libdcm1r`, dynamically linked to the application at run-time. There may be implementation-dependent procedure calls between `libdcm1r` and the DPCM, but this architecture isolates the application from such details.

NOTE—When the EDA application is statically linked, the link editor shall be told which external symbols need be resolved with dynamically-loaded code and how to locate that code. The specification of how to locate the code shall allow the application to choose from among multiple implementations of `libdcm1r`.

5.2.1 Global policies and conventions

The DPCM assumes a single locus of control and data understanding exists for callbacks. For example, if an application passes a pointer to a `port` structure as one argument in a call to the DPCM, the DPCM assumes it can pass that *same* pointer as an argument to a callback function.

In general, the DPCM shall not cache any design-specific data values between calls, since the DPCM cannot know when any such data might become invalid. The DPCM *can* cache within a single application call.

5.2.2 Flow of control

During execution, the flow of control within the DPCS moves between the application and the DPCM, subject to the following two constraints:

- a) There is no “abnormal” flow of control across its PI; that is, all function calls return control through the normal procedure-return mechanism.
- b) A DPCM never terminates the execution of the application to which it is linked.

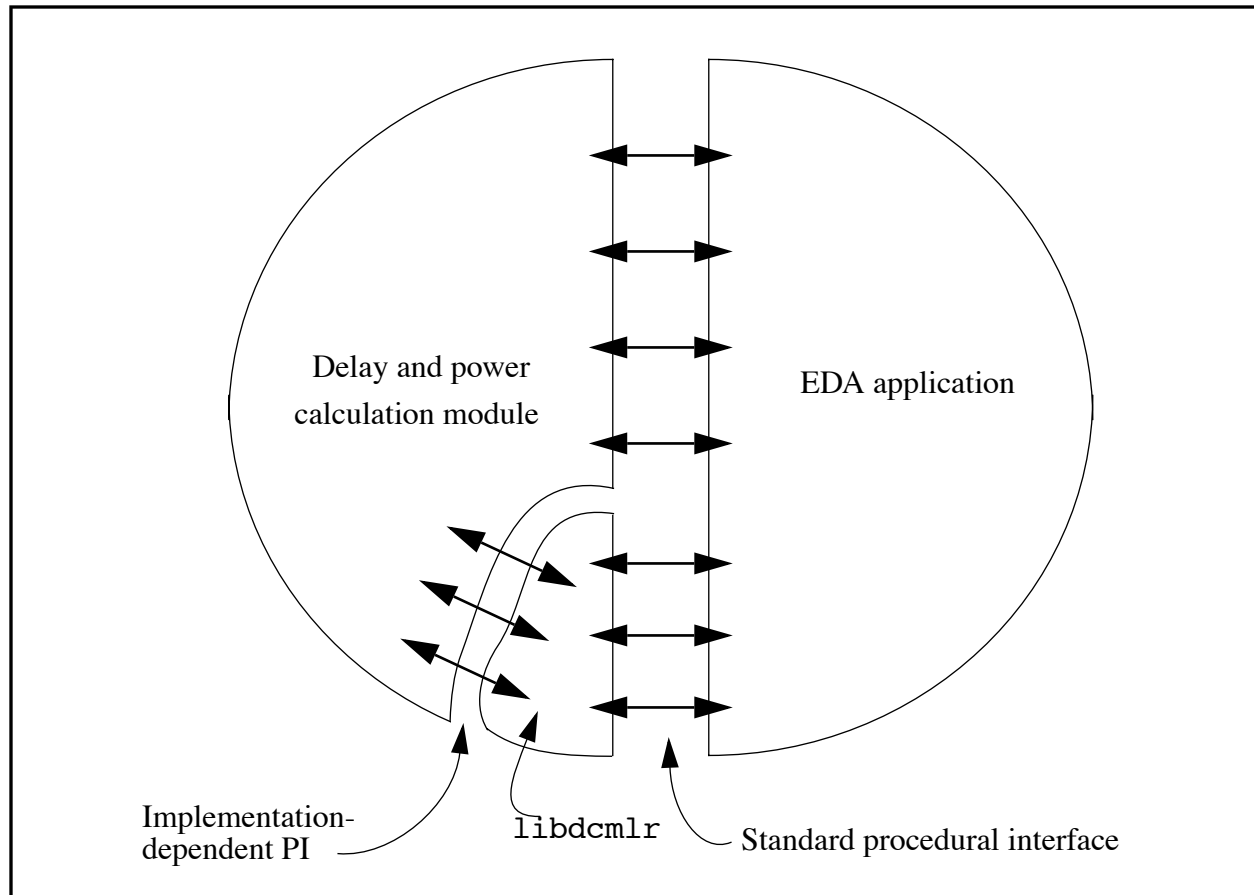


Figure 2—High-level DPCS architecture linkage structure

5.3 DPCM-application relationships

The DPCM has been designed to allow easy integration of dynamically linked executable modules for the purpose of delay and power calculation. The library developer decides how the components of the technology library should be implemented, then organizes the statements into logical groups called technology families and physical units called subrules. The run-time system assembles these units into a unified DPCM. The application then views the DPCM as a single subprogram from which it can request services.

5.3.1 Technology library

A technology library is a program written in DCL consisting of one or more subrules, each of which may contain references to other subrules (yet to be loaded). There is no hierarchical limit to the nesting of subrules within a technology library. Subrules may be segmented into technology families, which alters the way they are made available to the application (see 6.14).

5.3.2 Subrule

A subrule contains a combination of definitions and prototypes used to implement a portion of a technology definition. A subrule is a separate compilation unit and consists of one or more of the following independent components:

- C preprocessor directives
- Embedded C code

- DCL directives
- DCL atomic, model procedure, table manipulation, and library control statements

5.4 Interoperability

Consideration has been given to the possibility of having multiple implementations of supporting systems for this standard (compiler, run-time linker, and run-time library), and the desire to support at least a minimal level of interoperability among such implementations.

In particular, an application can concurrently access subrules constructed in multiple implementations by

- a) Limiting all subrules in a particular technology family to a single implementation.
- b) Appending an implementation-specific suffix to the *dcm* prefix for all PI calls with names beginning with *dcm*.
- c) Requiring the application to
 - 1) Track which technology families use which implementation.
 - 2) Use the appropriately-named PI calls.
 - 3) Maintain separate implementation-specific standard structures.

6. Delay calculation language (DCL)

This clause describes the delay modeling and calculation used in this standard. The formal syntax is described using the following backus naur form (BNF). For this clause, only following conventions are used:

- a) Lowercase words, some containing embedded underscores, are used to denote syntactic categories (terminals and nonterminals). For example:

literal_character_sequence

- b) Boldface words are used to denote reserved keywords, operators, and punctuation marks as a required part of the syntax. These words appear in a larger font for distinction. For example:

MODEL SLEW => ;

- c) The **::=** operator separates the two parts of a BNF syntax definition. The syntax category appears to the left of this operator and the syntax description appears to the right of the operator. For example, step d) shows five options for a *timing_mode_operator*.

- d) A vertical bar separates alternative items (use one only) unless it appears in boldface, in which case it stands for itself. For example:

timing_mode_operator ::= -> | <- | <-> | <-**X**-> | ->**X**<-

- e) Square brackets enclose optional items unless it appears in boldface, in which case it stands for itself. For example:

result_definition ::= **RESULT** ([result_sequence])

indicates *result_sequence* is an optional syntax item for *result_definition*, whereas

pin_range ::= pin_name [range_expression] pin_name

indicates square brackets are part of the syntax for *pin_range*.

- f) Braces enclose a repeated item unless it appears in boldface, in which case it stands for itself. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus, the following two rules are equivalent:

$$\begin{aligned} \text{comma_expression_list} &::= \text{expression } \{ , \text{expression } \} \\ \text{comma_expression_list} &::= \text{expression } | \text{comma_expression_list } , \text{expression} \end{aligned}$$

- g) Parenthesis enclose items within a group (use one only) unless it appears in boldface, in which case it stands for itself. In the following example:

$$\text{bus} ::= (\text{BUS } \text{physical_name } \{ \text{attribute } \} (\{ \text{net_ref } \} | \{ \text{pin_ref } \} \{ \text{node_ref } \}))$$

the first set of parenthesis are part of the syntax for a *bus* and the second set groups the items *net_ref* OR the combination of a *pin_ref* and *node_ref*.

- h) Angle brackets enclose items when no spacing is allowed between the items, such as within an *identifier*. In the following example:

$$\text{identifier} ::= \langle \text{identifier_first_char} \rangle \{ \langle \text{identifier_char} \rangle \}$$

the actual character(s) of the identifier cannot have any spacing.

- i) A hyphen (-) is used to denote a range. For example:

$$\text{identifier_first_letter} ::= \mathbf{a-z} | \mathbf{A-Z}$$

indicates the first letter of the identifier can be a lowercase letter (from a to z) or an uppercase letter (from A to Z).

- j) If the name of any category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *msb_constant_expression* and *lsb_constant_expression* are equivalent to *constant_expression*.

The main text uses *italicized* type when a term is being defined, and monospace font for examples, file names, and references to constants such as 0, 1, or x values.

6.1 Character set

The DCL character set shall be the same as that defined in section 5.2.1 of ISO/IEC 9899:1990, excluding trigraph sequences and multibyte characters. The alphabetic escape sequences such as '\n', representing new-line, shall be the same as those defined in section 5.2.2 of ISO/IEC 9899:1990.

6.2 Lexical elements

This subclause describes the lexical elements used to define the DCL syntax and semantics.

6.2.1 Whitespace

Whitespace is a contiguous sequence of one or more characters in the set: *space*, *horizontal_tab*, *newline*, *carriage_return*, *vertical_tab*, and *form_feed*.

6.2.2 Comments

A comment shall be either

- A character sequence that starts with `/*` and ends with the first occurrence of the character sequence `*/`. Within a comment, the character sequence `/*` shall not be recognized as starting a nested comment.
- A character sequence that starts with `//` and ends with the first occurrence of either the *newline* or *carriage_return* characters.

6.2.3 Tokens

The syntax for tokens in DCL is given in Syntax 1.

```
token ::=  
    keyword  
    | identifier  
    | double_quoted_character_sequence  
    | constant  
    | string_literal  
    | operator  
    | punctuation  
    | header_name
```

Syntax 1—Syntax for DCL token

Tokens other than *string_literal* and *double_quoted_character_sequence* shall not contain embedded *whitespace*.

6.2.3.1 Keyword

DCL reserves many tokens as *keywords* (see Table 1). These keywords shall be used only in the context as defined by DCL. No keyword shall be used as an identifier (see 6.2.3.2).

All keywords shall be case sensitive. Each keyword has two valid syntactic forms: one using only upper-case letters, and one using only lower-case letters. Table 1 shows only the upper-case syntactic form.

6.2.3.2 Identifier

An *identifier* is the name that represents a value, except within the `PATH`, `FROM`, `TO`, `BUS`, `TEST`, `INPUT`, `NODE`, and `OUTPUT` clauses, where it is treated as a literal string. An *identifier* is a sequence of one or more characters starting with an alphabetic character, followed by zero or more alphanumeric or underscore (`_`) characters.

The syntax for identifiers in DCL is given in Syntax 2.

An *identifier* is case sensitive. An *identifier* shall not be one of the following:

- A *keyword*
- A reserved word in the C or C++ languages (refer to ISO/IEC 9899:1990)
- Any character sequence beginning with the letters DCM in any mixture of case

Table 1—Keywords

ADD_ROW	FILE	REFERENCE
ANYIN	FILE_PATH	REFERENCE_EDGE
ANYOUT	FILTER	REFERENCE_EDGE_SCALAR
ASSIGN	FLOAT	REFERENCE_MODE
BIAS	FORWARD	REFERENCE_MODE_SCALAR
BLOCK	FROM	REFERENCE_POINT
BOTH	FROM_POINT	REFERENCE_SLEW
BUS	HOLD	REMOVAL
BY	IMPORT	REPLACE
CALC	IMPURE	REPEAT
CALC_MODE	INCONSISTENT	RESULT
CALC_MODE_SCALAR	INPUT	RISE
CALL	INPUT_PIN_COUNT	RULE_PATH
CELL	INTEGER	SETUP
CELL_DATA	INTERNAL	SETVAR
CELL_QUAL	KEY	SIGNAL
CGHT	LATE	SIGNAL_EDGE
CGPW	LATE_MODE	SIGNAL_EDGE_SCALAR
CGST	LATE_MODE_SCALAR	SIGNAL_MODE
CHECK	LATE_SLEW	SIGNAL_MODE_SCALAR
CHECKS	LOAD_TABLE	SIGNAL_POINT
CKTTYPE	METHOD	SIGNAL_SLEW
CLKFLG	METHODS	SINK_EDGE
COMPARE	MODEL	SINK_EDGE_SCALAR
COMPILATION_TIME_STAMP	MODEL_DOMAIN	SINK_MODE
CONSISTENT	MODEL_NAME	SINK_MODE_SCALAR
CONTROL_PARM	MODELPROC	SKEW
CPW	MONOLITHIC	SLEW
CST	NEW	SOURCE_EDGE
CYCLEADJ	NIL	SOURCE_EDGE_SCALAR
DATA	NOCHANGE	SOURCE_MODE
DEFAULT	NODE	SOURCE_MODE_SCALAR
DEFINES	NODE_COUNT	STATEMENTS
DELAY	NUMBER	STORE
DELETE_ROW	ONE_TO_Z	STRING
DHT	OPTIONAL	SUBMODEL
DO	OTHERWISE	SUBRULE
DOUBLE	OUTPUT	SUBRULES
DPW	OUTPUT_PIN_COUNT	SUPPRESS
DST	OVERRIDE	TABLE
DYNAMIC	PASSED	TABLEDEF
EARLY	PATH	TABLE_PATH
EARLY_MODE	PATH_DATA	TECH_FAMILY
EARLY_MODE_SCALAR	PATH_SEPARATOR	TERM
EARLY_SLEW	PHASE	TEST
EDGES	PIN	TEST_TYPE
END	PINLIST	TO
EXPORT	PROPAGATE	TO_POINT
EXPOSE	PROPERTIES	UNLOAD_TABLE
EXTERNAL	PROTOTYPE_RECORD	UNTIL
FALL	PURE	VAR
	QUALIFIERS	VOID
	RECOVERY	WHEN
		Z_TO_ONE
		Z_TO_ZERO
		ZERO_TO_Z

```
identifier ::= <identifier_first_character>{<identifier_character>}  
identifier_first_character ::= a-z | A-Z  
identifier_character ::= a-z | A-Z | 0-9 | _
```

Syntax 2—Syntax for DCL identifier

NOTE—It may be desirable to ensure that some *identifiers* are universally unique, so as to not collide when disparate library pieces from different companies are combined. Examples of such *identifiers* are **METHOD** names (see 6.12.7) and **TECH_FAMILY** names (see 6.14.1.1).

6.2.3.3 Double quoted character sequence

A *double_quoted_character_sequence* has the semantics of an *identifier*.

The syntax for double quoted characters in DCL is given in Syntax 3.

```
double_quoted_character_sequence ::= " literal_character_sequence "  
literal_character_sequence ::= <literal_character>{<literal_character>}  
literal_character ::= any character in the ASCII character set except  
double_quote, newline, or carriage_return
```

Syntax 3—Syntax for DCL double quoted character

6.2.3.4 Predefined references to standard structure fields

DCL defines a set of identifiers to reference fields in the *standard structure* (see 8.13). These identifiers shall be visible in all scopes. Table 2 lists the predefined identifiers.

6.2.3.5 Compiler generated predefined identifiers

The DCL compiler generates the values for the predefined identifiers shown in Table 3.

6.2.3.6 Constant

A *constant* is defined as either a floating point constant number or an integer constant number.

The syntax for constants in DCL is given in Syntax 4.

The *floating_constant* and *integer_constant* tokens have the same definition as in ISO/IEC 9899:1990.

6.2.3.6.1 Predefined constant NIL

NIL is a predefined constant of ISO C type `void *` with a value of `(void *) 0`, which shall indicate the **PINLIST**, **PIN**, **STRING**, **ARRAY**, or **VOID** has no value defined.

6.2.3.6.2 Edge type enumerations

The values for the **SINK_EDGE** and **SOURCE_EDGE** predefined identifiers are enumerations of the edge values as shown in Table 4. Use of these reserved words results in the corresponding string values in the generated code.

Table 2—DCL predefined references to standard structure fields

Predefined identifier	Data type	Set by DPCM or passed in by application	Description
BLOCK	PIN	Passed in during model search, passed in calculation.	The instance block identification in the design.
CALC_MODE	STRING	Passed in calculation.	Indicates the type of calculation associated with the current propagate segment calculation: bestCase, worstCase, or nominal.
CALC_MODE_SCALAR	INTEGER	Passed in calculation.	Indicates the type of calculation associated with the current propagate segment calculation: bestCase, worstCase, or nominal.
CELL	STRING	Passed in during model search, passed in calculation.	The cell of the circuit under calculation. The unique cell identification is the concatenation of CELL, CELL_QUAL and MODEL_DOMAIN (separated by a period).
CELL_DATA	VOID	Set during model search, passed in calculation.	Initial pointer to the caching system for cell-based store clauses.
CELL_QUAL	STRING	Passed in during model search, passed in calculation.	The cell of the circuit under calculation. The unique cell identification is the concatenation of CELL, CELL_QUAL and MODEL_DOMAIN (separated by a period).
CKTTYPE	STRING	Set by DPCM.	Reserved for library developer's use.
CLKFLG	STRING	Set during model search, passed in calculation.	Holds the clock flag to be assigned to this calculation.
CYCLEADJ	INTEGER	Set during model search.	Reserved for library developer's use.
EARLY_SLEW	FLOAT	Passed in calculation.	Holds the slew associated with the propagate segment for the delay or slew to be calculated. This slew represents the SLEW of the earliest signal to arrive at the segment for which the DELAY or SLEW is being calculated.
FROM_POINT	PIN	Set during model search, passed in calculation.	Holds the from point of the propagate segment for the delay or slew to be calculated.

Table 2—DCL predefined references to standard structure fields (continued)

Predefined identifier	Data type	Set by DPCM or passed in by application	Description
INPUT_PIN_COUNT	INTEGER	Passed in during model search.	Holds the number of input pins on the circuit currently being modeled or under calculation.
LATE_SLEW	FLOAT	Passed in calculation.	Holds the slew associated with the propagate segment for the delay or slew to be calculated. This represents the SLEW of the latest signal to arrive at the segment for which the DELAY or SLEW is being calculated.
MODEL_DOMAIN	STRING	Passed in during model search, passed in calculation.	The cell of the circuit under calculation. The unique cell identification is the concatenation of CELL , CELL_QUAL and MODEL_DOMAIN (separated by a period).
MODEL_NAME	STRING	Set during model search.	Holds the name of the modelproc that is currently in control.
NODE_COUNT	INTEGER	Passed in during model search.	Holds the number of nodes connected to the circuit currently being modeled or under calculation.
OUTPUT_PIN_COUNT	INTEGER	Passed in during model search.	Holds the number of output pins connected to the circuit currently under calculation or model build.
PATH	STRING	Set during model search, passed in calculation.	Holds the path or test statement name.
PATH_DATA	VOID	Set during model search, passed in calculation.	Holds the pointer to the path and pin base cache.
PHASE	STRING	Set by DPCM.	Gives access to phase. This is a combination of the SOURCE_EDGE and SINK_EDGE . When the SOURCE_EDGE and the SINK_EDGE are the same value, PHASE is set to I. If they are not the same value, PHASE is set to O.
REFERENCE_EDGE	STRING	Set during model search, passed in calculation.	Indicates whether the calculation is for the rising, falling, or both edges.
REFERENCE_EDGE_SCALAR	INTEGER	Set during model search, passed in calculation.	Indicates whether the calculation is for the rising, falling, or both edges.

Table 2—DCL predefined references to standard structure fields (continued)

Predefined identifier	Data type	Set by DPCM or passed in by application	Description
REFERENCE_MODE	STRING	Set during model search, passed in calculation.	Indicates whether the calculation is for early mode, late mode, or both.
REFERENCE_MODE_SCALAR	INTEGER	Set during model search, passed in calculation.	Indicates whether the calculation is for early mode, late mode, or both.
REFERENCE_POINT	PIN	Set during model search, passed in calculation.	Holds the from point of the propagate segment for the delay or slew to be calculated and is generally used in TEST and TABLEDEF statements.
REFERENCE_SLEW	FLOAT	Passed in calculation.	Holds the slew associated with the test segment for which the check is being analyzed. This slew represents the SLEW of the reference or “clock.” This allows the REFERENCE_SLEW to alter the calculations performed by the CHECK statement. The CHECK statement can adjust the BIAS by an amount which is a function of the reference’s slew.
SIGNAL_EDGE	STRING	Set during model search, passed in calculation.	Indicates whether the calculation represents the rising edge, falling edge, or both.
SIGNAL_EDGE_SCALAR	INTEGER	Set during model search, passed in calculation.	Indicates whether the calculation represents the rising edge, falling edge, or both.
SIGNAL_MODE	STRING	Set during model search, passed in calculation.	Indicates whether the calculation represents early mode, late mode, or both.
SIGNAL_MODE_SCALAR	INTEGER	Set during model search, passed in calculation.	Indicates whether the calculation represents early, mode, late mode, or both.
SIGNAL_POINT	PIN	Set during model search, passed in calculation.	Holds the to point of the propagate segment for the delay or slew to be calculated and is generally used in TEST and TABLEDEF statements. This is used during test computations.

Table 2—DCL predefined references to standard structure fields (continued)

Predefined identifier	Data type	Set by DPCM or passed in by application	Description
SIGNAL_SLEW	FLOAT	Passed in calculation.	Holds the slew associated with the test segment for which the check is being analyzed. This slew represents the SLEW of the signal or “data.” This allows the SIGNAL_SLEW to alter the CHECK statement's calculation. The BIAS computed by the CHECK statement can be a function of the signal's slew.
SINK_EDGE	STRING	Set during model search, passed in calculation.	Indicates whether the calculation represents the rising edge, falling edge, or both.
SINK_EDGE_SCALAR	INTEGER	Set during model search, passed in calculation.	Indicates whether the calculation represents the rising edge, falling edge, or both.
SINK_MODE	STRING	Set during model search, passed in calculation.	Indicates whether the calculation represents early mode, late mode, or both.
SINK_MODE_SCALAR	INTEGER	Set during model search, passed in calculation.	Indicates whether the calculation represents early mode, late mode, or both.
SOURCE_EDGE	STRING	Set during model search, passed in calculation.	Indicates whether the calculation is for the rising edge, falling edge, or both.
SOURCE_EDGE_SCALAR	INTEGER	Set during model search, passed in calculation.	Indicates whether the calculation is for the rising edge, falling edge, or both.
SOURCE_MODE	STRING	Set during model search, passed in calculation.	Indicates whether the calculation is for early mode, late mode, or both.
SOURCE_MODE_SCALAR	INTEGER	Set during model search, passed in calculation.	Indicates whether the calculation is for early mode, late mode, or both.
TO_POINT	PIN	Set during model search, passed in calculation.	Holds the to point of the propagate segment for which the delay or slew is to be calculated. It is used during delay and slew computations.

Table 3—DCL compiler generated predefined identifiers

Predefined identifier	Data type	Description
COMPILATION_TIME_STAMP	STRING	The time/date stamp when this rule was compiled.
CONTROL_PARM	STRING	Gives access to the CONTROL_PARM value specified in the SUBRULE statement which loaded this rule.
RULE_PATH	STRING	Gives access to the RULE_PATH value specified in the SUBRULE statement which loaded this rule.
TABLE_PATH	STRING	Gives access to the TABLE_PATH value specified in the SUBRULE statement which loaded this rule.

constant ::= floating_constant | integer_constant

Syntax 4—Syntax for DCL constant

The remaining edge types defined in a DCM_EdgeTypes structure—SAME, BOTH, ALL, COMPLIMENT, and TERMINATEBOTH—shall be considered illegal enumeration values for SINK_EDGE and SOURCE_EDGE.

Table 4—Edge types and conversions

Keyword	Definition	String value	enum value
FALL	The FALL edge means a signal transitions from a high level to a low level. The levels in DCL are arbitrary as there are no specified reference levels since they are implied in the calculation.	'F'	1
ONE_TO_Z	ONE_TO_Z means the signal is transitioning from a high level to a high impedance state.	'1Z'	7
RISE	The RISE edge means a signal transitions from a low level to a high level. The levels in DCL are arbitrary as there are no specified reference levels since they are implied in the calculation.	'R'	0
TERM	TERM means the edge terminates and does not propagate.	'T'	5
Z_TO_ONE	Z_TO_ONE means the signal is transitioning from a high impedance state to a high level.	'Z1'	8
ZERO_TO_Z	ZERO_TO_Z means the signal is transitioning from a low level to a high impedance state.	'0Z'	9
Z_TO_ZERO	Z_TO_ZERO means the signal is transitioning from a high impedance state to a low level.	'Z0'	10

NOTE—See 8.11.7 for DCM_EdgeTypes.

6.2.3.6.3 Propagation type enumerations

SINK_MODE and SOURCE_MODE are converted to enumerations as shown in Table 5.

Table 5—Propagation mode conversions

Keyword	Definition	String value	enum value
EARLY	The very first edge that propagates through a given cone of logic.	'E'	0
LATE	The very last edge that propagates through a given cone of logic.	'L'	1

6.2.3.6.4 Calculation mode enumeration

CALC_MODE values are converted to enumerations as shown in Table 6.

Table 6—Calculation mode conversions

Definition	String value	enum value
Best case	'B'	0
Worst case	'W'	1
Nominal case	'N'	2

6.2.3.6.5 Test type enumeration

TEST_TYPE values are converted enumerations as shown in Table 7.

6.2.3.7 String literal

A *literal_character* is any member of the character set except the single-quote ('), backslash (\), or *newline_character*. Each of these restricted characters may be present in a single quoted string if it is preceded by a backslash.

The syntax for string literals in DCL is given in Syntax 5.

string_literals follow the same semantics as ISO C string literals (see section 6.1.4 of ISO/IEC 9899:1990).

6.2.3.8 Operators

The syntax for DCL operators is given in Syntax 6.

6.2.3.9 Punctuator

The syntax for punctuators is given in Syntax 7.

The punctuators (), [], %{ }%, and { } shall occur in balanced pairs.

Table 7—TEST_TYPE conversions

Keyword	Definition	enum Value
CGHT	Clock gating hold test	7
CGPW	Clock gating pulse width test	6
CGST	Clock gating setup test	8
CPW	Clock pulse width test	2
CST	Clock separation test	3
DHT	Data hold test	10
DPW	Data pulse width test	4
DST	Data setup test	5
HOLD	Hold test	1
NOCHANGE	No change test	14
RECOVERY	Recovery test	11
REMOVAL	Removal test	12
SETUP	Setup test	0
SKEW	Skew test	13

```
string_literal ::= ' {< literal_character> }'
```

Syntax 5—Syntax for DCL string literal

```
operator ::=
    $ | ^ | * | / | + | = | ** | || | & & | ! | == | != | > | < | >= | <=
    | -> | <- | <-> | ->X<- | <-X-> | NEW | :: | .
```

Syntax 6—Syntax for DCL operator

```
punctuator ::= ( | ) | [ | ] | { | } | , | ; | : | . | # | & | < | > | % | { | } | % | ' | "
```

Syntax 7—Syntax for DCL punctuator**6.2.3.10 Name**

A *name* is either an *identifier* or a *double_quoted_character_sequence*. The use of these punctuators is incorporated within the subsequent syntax charts in this clause.

6.2.4 Header names

Header name preprocessing tokens shall only appear with a `#include` preprocessing directive. The header name shall be defined the same as in section 6.1.7 of ISO/IEC 9899:1990.

6.2.5 Preprocessing directives

DCL preprocessing directives are exactly the set defined in section 6.8 of ISO/IEC 9899:1990 and have the same semantics.

6.3 Name spaces of identifiers

If more than one declaration of a particular identifier is visible at any point in the translation unit, the syntactic context determines proper reference to different entities. Thus, there are separate name spaces for various categories of identifiers, as follows:

- Identifiers declared in **RESULT** clauses.
- Identifiers naming methods and technology families.
- All other identifiers.

6.4 Storage durations of objects

An object has a *storage duration* that determines its lifetime. There are two storage durations: *static* and *automatic*.

An object defined by an **ASSIGN** statement or whose identifier is declared with external or internal linkage shall have *static storage duration*. For such an object, storage shall be reserved prior to program start-up. The object shall exist and retain its last-stored value throughout the execution of the entire program.

An object defined by a **STORE** clause shall have *static storage duration*. For such an object, storage shall be reserved when the enclosing **MODELPROC** statement is evaluated. The object shall exist and retain its last-stored value throughout the execution of the entire program.

All non-array objects whose identifiers are not defined by a **STORE** clause or an **ASSIGN** statement shall have *automatic storage duration*. For such objects, their storage shall be allocated when the defining statement is called and shall persist until the calling statement goes out of scope.

For all array objects, their storage shall be allocated by the **NEW** operator and shall persist (while in scope within the DPCM) at least until control reaches the application or while the application or the DPCM has locked the array (see 8.4.1).

6.5 Scope of identifiers

An *identifier* shall be visible (i.e., can be used) only within a region of program text referred to as its *scope*. There are ten different types of scope, as follows:

- Global
- **TECH_FAMILY**
- Subrule
- Statement-prototype
- Statement
- Modeling procedure
- **STATEMENTS**
- **RESULT**
- **PASSED**
- Discrete

When lexically identical identifiers exist in the same name space, identifiers in outer scopes shall be hidden until the inner scope terminates. The DCL scope hierarchy is

```

global
    TECH_FAMILY
        subrule
            statement-prototype
            statement
                RESULT
                    PASSED
                        discrete
                            modeling procedure
                                STATEMENTS
                                    statement

```

The scope of an *identifier* shall start where the *identifier* is first declared and extend to the end of the scope in which it was declared. An *identifier* shall be declared before it is referenced. Multiple MODELPROC statements of the same name shall not occur within the same TECH_FAMILY.

6.6 Linkages of identifiers

A statement *identifier* declared in different scopes can be made to refer to the same object or statement by a process called *linkage*. There are three subrule scope options affecting linkage: EXPORT, IMPORT, and FORWARD. Expose chaining also defines how EXPOSE identifiers are linked within a TECH_FAMILY's scope. (See 8.8.1.1 for more details.)

6.6.1 EXPORT

A statement *identifier* shall be made visible outside its declared subrule scope and within the same TECH_FAMILY by using the EXPORT option on a statement definition.

6.6.2 IMPORT

A statement *identifier* EXPORTed in one subrule scope shall be made visible within another subrule scope and within the same TECH_FAMILY by using the IMPORT option on a statement prototype.

6.6.3 FORWARD

A statement *identifier* referenced in a subrule scope before its definition (within the same scope) shall be made visible by using the FORWARD option on a statement prototype.

6.6.4 Chaining of EXPOSE identifiers

EXPOSE statement identifiers with the same name in separate subrules within the same TECH_FAMILY are chained together at run-time. The application and DPCM are presented with a single EXPOSE entry point for this identifier. The first EXPOSE statement in a chain is always used as the reference to any EXPOSE statement within a particular subrule (even if there is an EXPOSE statement defined within the subrule containing the reference).

6.7 DCL data types

DCL supports three categories of data types: *native*, *derived*, and *array*. The native data types are defined in terms of the ISO C data types. See 8.11.6 for details of the mapping between DCL and ISO C data types.

6.7.1 Native data types

The syntax for native data types in DCL is given in Syntax 8.

```
native_type ::= mathematical_type | pointer_data_type
```

Syntax 8—Syntax for DCL native data type

6.7.1.1 Mathematical calculation data types

The syntax for calculation data types in DCL is given in Syntax 9.

```
mathematical_type ::= NUMBER | DOUBLE | FLOAT | INTEGER
```

Syntax 9—Syntax for DCL calculation data type

6.7.1.2 Pointer data types

The syntax for pointer data types in DCL is given in Syntax 10.

```
pointer_data_type ::= STRING | PIN | PINLIST | VOID
```

Syntax 10—Syntax for DCL pointer data type

6.7.2 Array types

The syntax for array types in DCL is given in Syntax 11.

```
native_array_type ::= [ mathematical_type | STRING | PIN | VOID ] [ dimension_list ]  
dimension_list ::= fixed_dimension_list | variable_dimension_list  
fixed_dimension_list ::= scalar { , scalar }  
variable_dimension_list ::= * { , * }
```

Syntax 11—Syntax for DCL array type

An array type describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*. Array types are modeled by their element type, their number of dimensions, and the number of elements in each dimension of the array.

An array dimension may have type *fixed* or *variable*. The number of elements in a *fixed* dimension shall be specified when the array is declared and shall not vary during program execution. The number of elements in a *variable* dimension shall not be specified when the array is declared and may vary during program execution.

The following are true for all array types:

- Arrays can have an arbitrary number of dimensions up to a limit of 255. For any particular array, all of its dimensions shall have the same type—*fixed* or *variable*.

- The length of each dimension in a multidimensional array shall not depend on the index used for any of the other dimensions of that array—“ragged” arrays shall not be supported.
- Array indices shall start with zero (0) in each dimension.
- Array types shall be passed and returned as pointers to a contiguous memory region. The array data shall be organized as defined by ISO/IEC 9899:1990.

Array types shall be constant unless declared to be VAR. A VAR array may have its elements changed during program execution. An array type that has the VAR declaration may be passed as an argument to a statement requiring a constant array. A constant array type shall not be passed as an argument to a statement requiring a VAR array type.

The VAR array modifier is defined in Table 8.

Table 8—VAR array modifier

Symbol	Definition
VAR, var	Indicates (to the compiler) the array field is modifiable by any variable that references the array. Without the var array modifier, arrays are constant and cannot be modified in any way.

Example

```
FORWARD CALC(name):
RESULT(INTEGER[*,*]: constIntArray &
      NUMBER VAR [*]: varNumVec);
```

The VAR array modifier indicates array elements may be altered by statements other than the statement that created the array. Assignments for non-var to non-var, var to var, or var to non-var are effected through pointer manipulation; no copying of array values is performed.

6.7.3 Derived data types

The syntax for derived data types is given in Syntax 12.

```
derived_type ::= result_type | statement_type

result_type ::=
    assign_statement_name
  | calc_statement_name
  | expose_statement_name
  | external_statement_name
  | internal_statement_name
  | tabledef_statement_name

statement_type ::= result_type ( )
```

Syntax 12—Syntax for DCL derived data type

6.7.3.1 Result types

RESULT clauses in statements and DATA clauses in TABLEDEFs define derived data types which shall be analogous to structures in ISO/IEC 9899:1990. The name of the associated statement or TABLE can be used as the name of the derived data type. RESULT type field ordering is defined in 6.12.1.2.1 and 6.12.1.2.2.

Example

```
calc(resultType):    passed(integer:x)
                    result(double:fp=2*x+3 & integer:x+1);
calc(example):      passed(resultType:x)
                    result(double:x.fp*4 - 2*x);
calc(exercise):     result(double:example(resultType(5)));
```

6.7.3.2 Statement types

A statement name followed by () defines a derived data type. Any statement having the same PASSED arguments (number, order, and type) and RESULT structure matches this derived type. This enables the passing of statement pointers into other statements.

Example

exercise() passes statementType1() to example() as a derived type:

```
calc(statementType1): passed(integer:x)
                     result(integer:resType(x)+ 3*x);
calc(statementType2): passed(integer:z)
                     result(integer:resType(z)+6*z+ 1);
calc(example):        passed(statementType1(): y)
                     result(integer: y(5));
calc(exercise):       result(integer:example(statementType1)+
                           example(statementType2));
```

6.8 Type conversions

Type conversions shall be performed based on

- Changing the type of an argument to the type of the expected parameter, or
- Changing the type of an expression term to the type of other terms in the same expression.

Only the conversions enumerated in the following subclauses are valid in DCL.

6.8.1 Implicit conversions

The following implicit type conversions shall be performed:

- INTEGER to DOUBLE or FLOAT
An INTEGER value shall be converted to DOUBLE (FLOAT) when the expected parameter or expression term has type DOUBLE (FLOAT). An INTEGER value shall be converted to DOUBLE when a division operator is present in the expression.
- FLOAT to DOUBLE
A FLOAT value shall be converted to DOUBLE when the expression term has type DOUBLE.

- **PIN to STRING**
A **PIN** value shall be converted to **STRING** when the expected parameter or expression term has type **STRING**.
- **native_array_type, pointer_data_type or derived_type to VOID**
A **native_array_type, pointer_data_type or derived_type** value shall be converted to **VOID** when the expected parameter or expression term has type **VOID**.

6.8.2 Explicit conversions

The following explicit type conversions shall be performed:

- **FIXED-dimension array to VARIABLE-dimension array**
A **FIXED-dimension array** shall be converted to a **VARIABLE-dimension array** when a fixed dimension array is assigned to a variable dimension array of the same dimensionality and data type.
- **DERIVED data type to DERIVED data type**
A **DERIVED data type** shall be converted to an equivalent **DERIVED data type** when the expected parameter has the same structure (order, number, and native types) as the argument.

Example

```
CALC(resType):      passed(integer:x)
                    result(double:fp = 2*x+3 & integer:x+1);
CALC(resType2):     passed(integer:x)
                    result(double:y=2**x & integer:z=x-1);
CALC(example):      passed(resType:x)
                    result(double:x.fp * 4 - 2 * x);
CALC(exercise):     result(double:example(resType(5))
                    +example(resType2(3)));
```

6.9 Operators

This subclause details the operators used in DCL.

6.9.1 String prefix operator

This subclause details the use of string prefix operators.

6.9.1.1 Explicit string prefix operator

The *explicit string prefix operator* is a unary operator that modifies the semantics of logical comparisons between string operands. This operator can be used with string operands within logical expressions.

The explicit string prefix operator is the * (asterisk) character. This operator shall occur as a separate token that precedes a string literal or an *identifier* having a string value.

6.9.1.2 Embedded string prefix operator

The *embedded string prefix operator* is a unary operator that modifies the semantics of logical comparisons between string operands. This operator can be used with string operands within logical expressions, table qualifiers, and table references.

The embedded string prefix operator is the * (asterisk) character. This operator shall occur as the first character of the string literal, the first character of a name, or the first character of the value of an identifier. To disable this special meaning of an asterisk, it shall be preceded with a backslash character (\).

6.9.1.3 String prefix semantics

The semantics of the explicit and embedded string prefix operator are identical. When the string prefix operator is present, the logical comparison shall be performed considering only a subset of the characters in the strings. The comparison shall start with the first character after the prefix operator, if present, in each string and shall proceed for the length of the string with the string prefix operator. If both strings have the prefix operator, then the comparison shall proceed for the length of the shorter string.

If the explicit string prefix operator is used and its operand contains a leading asterisk, this leading asterisk shall *not* be considered an operator; it shall be treated as the first character of the string.

6.9.2 Assignment operator

For native types, except arrays, the *assignment operator* (=) shall assign the expression value on the right to a variable on the left. For arrays, this operator shall make the variable on the left (which shall be an array name) a *reference* to the array on the right.

6.9.3 New operator

The *new operator* (NEW) shall be used to create memory space for arrays. NEW is used when a new instance of an array is needed. After the NEW operator creates the space for an array, the values of the individual members shall be undefined.

6.9.4 SCOPE operator

The scoping prefix PATH_DATA:: or CELL_DATA:: determines what *action statement* is associated with a method (see 6.12.7), and what the value of a store variable is, based on the value of the corresponding pre-defined identifier PATH_DATA or CELL_DATA (see Syntax 16).

6.9.5 Purity operator

A reference to an impure statement shall be treated as PURE if the reference is preceded by the *purity operator* (^) (see 6.12.2.1). The purity operator is defined in Table 9.

Table 9—Purity operator

Symbol	Definition
$\wedge$	The unary purity operator declares the statement reference immediately following it to be PURE, thereby overriding any default assumptions based on the optimization rules. References prefixed with the purity operator shall be optimized as a PURE reference.

6.9.6 Timing propagation

Static timing includes the need to resolve timing when two or more input signals converge at a node. For each signal, a time period (window) can be defined based on the earliest and latest possible times when that

signal can arrive at the node. The resolution process at a node determines a window for the node's output signal based on the windows of the input signals for that node.

DCL supports five ways (modes) to perform this resolution, defined in Table 10 and used in PROPAGATE, EDGE, and TEST clauses.

Table 10—Timing resolution modes

Symbol	Definition
->	early(output)= earliest of all early(input) late(output)= early(output)
<-	late(output)= latest of all late(input) early(output)= late(output)
<->	early(output)= earliest of all early(input) late(output)= latest of all late(input)
->X<-	early(output)= latest of all early(input) late(output)= latest of all late(input)
<-X->	early(output)= earliest of all early(input) late(output)= earliest of all late(input)

6.9.7 Timing checks

Static timing includes the need to compare the windows of different signals at different nodes. One signal is always chosen as the reference. The comparison done is determined by a combination of test type. See 6.2.3.6.5, 6.9.8, and 6.15.4.4.3.

DCL supports three ways to perform this comparison, as shown in Table 11.

Table 11—Test mode operators

Symbol	Definition
->	Signal must arrive later than reference + bias Bias shall be computed using early signal and late reference values.
<-	Signal must arrive earlier than reference – bias Bias shall be computed using late signal and early reference values.
<->	Signal must arrive earlier than reference – offset1, and signal must arrive later than reference + offset2

There are four possible window-edge comparisons; the early or late edge of the signal window can be compared with the early or late edge of the reference window. The test mode operand refines the window comparison by specifying which ends of the windows to compare, along with some additional semantics.

Typically, two windows are maintained for each signal, corresponding to rising and falling transitions. The EDGES clause selects which window is used for the comparison.

6.9.8 Test mode operators

This subclause details the use of test mode operators.

6.9.8.1 Clock gating hold test mode operator (CGHT)

The clock gating hold test mode operator (CGHT) is similar to the HOLD test mode operator except the reference edge shall be from a clock pin and the signal shall be from a data pin.

6.9.8.2 CLock gating pulse width test mode operator (CGPW)

The clock gating pulse width test mode operator (CGPW) is similar to the CPW mode operator except the pin specified for the test shall be from a clock gate (logic signal).

6.9.8.3 Clock gating setup test mode operator (CGST)

The clock gating setup test mode operator (CGST) is similar to the SETUP test mode operator except the reference edge shall be from a data pin and the signal shall be from a clock pin.

6.9.8.4 Clock pulse width test mode operator (CPW)

The clock pulse width test mode operator (CPW) specifies the edge identified as the signal shall be offset from the edge identified by the reference. The amount of the offset is at least as large as the bias value. The edge identified as the signal shall occur before the edge identified as the reference for late mode and positive bias values. The edge identified as the signal shall arrive after the edge identified as the reference for early mode and positive bias values. Both edges specified for this test shall be from the same pin and a clock.

6.9.8.5 Clock separation test mode operator (CST)

The clock separation test mode operator (CST) specifies the edge identified as the signal shall be offset from the edge identified as the reference. The amount of the offset is at least as large as the bias value. The edge identified as the signal shall occur before the edge identified as the reference for late mode and positive bias values. The edge identified as the signal shall arrive after the edge identified as the reference for early mode and positive bias values. The edges specified for this test mode operator shall be from different clock pins.

6.9.8.6 Data hold test mode operator (DHT)

The data hold test mode operator (DHT) specifies the separation of two data signals on the same cycle. The DHT specifies an edge of one data signal against another edge of another data signal. The constraint is calculated by the CHECKS clause. As with other hold type test mode operators, this test shall ensure the SIGNAL_EDGE comes after the REFERENCE_EDGE. The difference between this test mode operator and DST (see 6.9.8.8) is DST implies there is a cycle adjustment made to the reference signal before the data separation test is performed. There is no cycle adjust added to the reference in DHT.

6.9.8.7 Data pulse width mode operator (DPW)

The data pulse width test mode operator (DPW) specifies the edge identified as the signal shall be offset from the edge identified by the reference. The amount of the offset is at least as large as the bias value. The edge identified as the signal shall occur before the edge identified as the reference for late mode and positive bias values. The edge identified as the signal shall arrive after the edge identified as the reference for early mode and positive bias values. Both edges specified for this test mode operator shall be from the same pin and not a clock.

6.9.8.8 DST

The data setup test mode operator (DST) is used to determine the offset between two data signals. This separation is established by the CHECKS clause. The specified edge on the data pin specified as the SIGNAL_EDGE shall arrive before the edge specified on the data pin specified as the REFERENCE_EDGE for positive values of the offset.

6.9.8.9 HOLD

The HOLD test mode operator specifies (for positive bias values) the earliest edge identified as the signal shall arrive after the latest edge identified as the reference. The HOLD test mode operator shall be used in conjunction with early mode or following the both modes operator. The reference edge shall be from a clock pin.

6.9.8.10 NOCHANGE

The NOCHANGE test mode operator specifies the edge identified as the signal does not change during the duration of the setup period, the entire pulse width, and the hold period. The edge identified as the reference represents the earliest edge of the clock. The termination of the pulse width period is the opposite edge of the reference. The edge identified as the signal represents the earliest edge of the data. The nochange test mode operator requires the use of the combined early/late mode operator. There are two bias values for this test.

The bias value preceding the combined early late mode operator determines the period preceding the reference edge. The bias value following the combined early/late operator determines the period following the opposite edge of the reference. The reference edge shall be from a clock or control pin and the signal edges shall be from a logic pin.

6.9.8.11 RECOVERY

The RECOVERY test mode operator specifies the latest inactive edge identified as the SIGNAL_EDGE shall arrive before the earliest edge identified as the REFERENCE_EDGE. The bias value shall be positive. The SIGNAL_EDGE pin shall be from a control pin. The recovery test mode operator shall be used in conjunction with the late test mode operator or preceding the both modes operators. The REFERENCE_EDGE shall be from a clock pin.

6.9.8.12 REMOVAL

The REMOVAL test mode operator specifies the earliest inactive edge identified as the SIGNAL shall arrive after the latest edge identified as the reference. The bias value shall be positive. The SIGNAL pin shall be from a control pin. The removal test mode operator shall be used in conjunction with the late test mode operator or following the both modes operators. The reference edge shall be from a clock pin.

6.9.8.13 SETUP

The SETUP test mode operator specifies (for positive bias values) the latest edge identified as the SIGNAL shall arrive before the earliest edge identified as the reference. The SETUP test mode operator shall be used in conjunction with the late mode or preceding the both modes operators. The reference edge shall be from a clock pin.

6.9.8.14 SKEW

The SKEW test mode operator specifies the edge identified as the source shall occur within a window of time either before or after the edge identified as the reference. The bias sets the magnitude of the window. The bias value shall be positive. The use of this test mode operator with the early test mode operator indicates

that the signal can occur after the reference up to the bias limit. The use of this test mode operator with the late mode operator indicates the signal may occur before the reference by an amount up to the bias value. Both the signal and reference shall be from clock pins and only the early mode or late mode operators shall be used.

6.10 Expressions

An expression is a sequence of operators and operands that

- Specifies the computation of a value.
- Designates an object or a function.
- Generates side effects.
- Performs a combination of these.

The order of evaluation of subexpressions and the order in which side effects take place are unspecified, except as indicated by the syntax or when explicitly specified.

6.10.1 Array subscripting

The syntax for array subscripting is given in Syntax 13.

<pre>array_index ::= [integer_comma_expression_list] comma_expression_list ::= expression { , expression }</pre>
--

Syntax 13—Syntax for array subscripting

A reference to an array element shall be made using the array name followed by a nonempty list of subscript expressions, separated by commas and surrounded by square brackets. Each subscript expression shall be of integer data type and evaluate to an integer value.

NOTE—The ISO C array reference `a[b][c][d]` is expressed as `a[b,c,d]` in DCL.

6.10.2 Statement calls

This subclause describes statement calls in DCL.

6.10.2.1 General syntax

The syntax for statement calls is given in Syntax 14.

<pre>statement_call ::= statement_name ([comma_expression_list]) statement_reference ::= statement_call [array_index] . field_reference</pre>

Syntax 14—Syntax for statement call

Arguments passed to a *statement* shall be read-only within that statement, with the sole exception of VAR arrays (see Table 8), which shall also be writable by the *statement*.

6.10.2.2 Built-in function calls

The syntax for built-in function calls is given in Syntax 15.

```
built-in_function_call ::= built-in_name ( [comma_expression_list ] )
```

Syntax 15—Syntax for built-in function call

6.10.2.3 Method statement calls

The syntax for method statement calls is given in Syntax 16.

```
method_statement_call ::= method_statement_name ( )  
method_statement_reference ::=  
    method_statement_call [ array_index | . field_reference ]  
method_call ::=  
    method_statement_reference  
| PATH_DATA :: method_statement_reference  
| CELL_DATA :: method_statement_reference
```

Syntax 16—Syntax for method statement call

A method statement call made without specifying **CELL_DATA ::** scope shall default to a method statement call with the **PATH_DATA ::** scope.

6.10.3 Assign variable reference

The syntax for assign variable references is given in Syntax 17.

```
assign_variable_reference ::=  
    assign_statement_name [ array_index | . field_reference ]  
  
field_reference ::= result_field_name [ array_index ]
```

Syntax 17—Syntax for assign variable reference

The value referenced shall be the last calculation for the named **ASSIGN** statement.

6.10.4 Store variable reference

The syntax for store variable references is given in Syntax 18.

The value referenced shall be that stored during model elaboration for the current **PATH_DATA** or **CELL_DATA**. If neither the **PATH_DATA ::** nor **CELL_DATA ::** scope operators are specified, use of **PATH_DATA ::** shall be assumed.

6.10.5 Mathematical expressions

This subclause details the mathematical expressions allowed in DCL and gives their syntax (see Syntax 19).

```

store_variable_reference ::= scoped_variable_reference | slot_variable_reference

scoped_variable_reference ::=
    store_reference
    | PATH_DATA :: store_reference
    | CELL_DATA :: store_reference

store_reference ::=
    recallable_statement_name [ array_index | . field_reference ]

recallable_statement_name ::=
    calc_statement_name
    | internal_statement_name
    | external_statement_name
    | expose_statement_name
    | tabledef_statement_name

slot_variable_reference ::= array_index store_reference

```

Syntax 18—Syntax for store variable reference

```

expression ::=
    constant
    | identifier
    | string_literal
    | discrete_expression
    | statement_reference
    | assign_variable_reference
    | store_variable_reference
    | method_call
    | built-in_function_call
    | c_statement_reference
    | variable_reference
    | + expression
    | expression + expression
    | - expression
    | expression - expression
    | expression * expression
    | expression / expression
    | expression ** expression
    | ( expression )

variable_reference ::= expr_variable [ array_index | . field_reference ]

expr_variable ::=
    assign_statement_name
    | passed_argument_name
    | result_field_name
    | predefined_variable_name

```

Syntax 19—Syntax for expression

6.10.5.1 Mathematical operators

The mathematical operators are defined in Table 12.

6.10.5.2 Discrete math expression

Discrete math represents sums or products of expressions. Discrete math expressions compute the sum of terms or the product of terms. The resulting type of the discrete expression is that of the *discrete_expression_body*.

Table 12—Mathematical operators

Symbol	Definition
*	Multiply
/	Divide
+	Add, unary plus
-	Subtract, unary minus
**	Exponentiation

The syntax for discrete math expressions is given in Syntax 20.

```

discrete_expression ::=
    discrete_declaration discrete_operator { discrete_expression_body }

discrete_declaration ::= [ pin_list_discrete ] | [ integer_discrete ]

pin_list_discrete ::= PINLIST : loop_variable_name = pinlist_expression

integer_discrete ::= discrete_bounds | discrete_bounds BY integer_expression

discrete_bounds ::= discrete_start discrete_end

discrete_start ::= INTEGER : loop_variable_name = integer_expression

discrete_end ::= TO integer_expression

discrete_operator ::= + | *

discrete_expression_body ::= expression

```

Syntax 20—Syntax for discrete math expression

The *discrete_operator* represents the type of discrete operation being performed

- * Indicates a product operation on the *loop_body* expression and
- + Indicates a summation operation on the *loop_body* expression.

Both the *integer_discrete* **BY** expression and the *discrete_end* expression shall be evaluated exactly once, after the *discrete_start* expression has been evaluated.

6.10.5.2.1 INTEGER discrete

The integer discrete expression indicates the loop variable has type **INTEGER** and can be incremented or decremented only by an integer value.

6.10.5.2.2 PINLIST discrete

The pinlist discrete expression indicates the loop variable has type **PINLIST** and is stepped though the list of PINs in the **PINLIST**.

6.10.6 Logical expressions and operators

This subclause details the logical expressions and logical operators allowed in DCL.

The syntax for logical expressions is given in Syntax 21.

```
logical_expression ::=
    prefix_expression == prefix_expression
  | prefix_expression != prefix_expression
  | prefix_expression >= prefix_expression
  | prefix_expression <= prefix_expression
  | prefix_expression < prefix_expression
  | prefix_expression > prefix_expression
  | ! logical_expression
  | ( logical_expression )
  | logical_expression && logical_expression
  | logical_expression || logical_expression

prefix_expression ::= * string_expression | expression
```

Syntax 21—Syntax for logical expression

The DCL logical operators are defined in Table 13.

Table 13—Logical operators

Symbol	Definition
	Or
&&	And
!	Not
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

6.10.7 Pin range

A pin range expression represents one or more input pins, or one or more output pins on a circuit.

6.10.7.1 Pin range syntax

The syntax for a pin range is given in Syntax 22.

NOTE—The use of angle brackets (<>) or square brackets ([]) affects the expansion of the pin range (see 6.10.7.2).