

# 1450.1™

## IEEE Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450™1999) for Semiconductor Design Environments

---

### IEEE Computer Society

Sponsored by the  
Test Technology Standards Committee



3 Park Avenue, New York, NY 10016-5997, USA

30 September 2005  
Print: SH95344  
PDF: SS95344



# IEEE Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450<sup>TM</sup>-1999) for Semiconductor Design Environments

Sponsor

**Test Technology Standards Committee  
of the  
IEEE Computer Society**

Approved 9 June 2005

Reaffirmed 16 June 2011

**IEEE-SA Standards Board**

Approved 17 November 2005

Reaffirmed 25 July 2012

**American National Standards Institute**

**Abstract:** Standard Test Interface Language (STIL) provides an interface between digital test generation tools and test equipment. Extensions to the test interface language (contained in this standard) are defined that (1) facilitate the use of the language in the design environment and (2) facilitate the use of the language for large designs encompassing subdesigns with reusable patterns.

**Keywords:** advanced scan architecture, core, environment, fail feedback, lockstep, parallel patterns, parameterized data, pattern tiling, pragma, signal variable, system on chip (SoC), test protocol

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

Copyright © 2005 by the Institute of Electrical and Electronics Engineers, Inc.  
All rights reserved. Published 30 September 2005. Printed in the United States of America.

IEEE is a registered trademark in the U.S. Patent & Trademark Office, owned by the Institute of Electrical and Electronics Engineers, Incorporated.

Print: ISBN 0-7381-4732-X SH95344  
PDF: ISBN 0-7381-4733-8 SS95344

*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.*

Authorized licensed use limited to: Micronas GmbH. Downloaded on July 17, 2018 at 08:38:16 UTC from IEEE Xplore. Restrictions apply.

**IEEE Standards** documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. The IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While the IEEE administers the process and establishes rules to promote fairness in the consensus development process, the IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

Use of an IEEE Standard is wholly voluntary. The IEEE disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other IEEE Standard document.

The IEEE does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained herein is free from patent infringement. IEEE Standards documents are supplied “**AS IS.**”

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, or every ten years for stabilization. When a document is more than five years old and has not been reaffirmed, or more than ten years old and has not been stabilized, 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.

In publishing and making this document available, the IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is the IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other IEEE Standards document, should rely upon his or her independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

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 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. A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered the official position of IEEE or any of its committees and shall not be considered to be, nor be relied upon as, a formal interpretation of the IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position, explanation, or interpretation of the IEEE.

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. Recommendations to change the status of a stabilized standard should include a rationale as to why a revision or withdrawal is required. Comments and recommendations on standards, and requests for interpretations should be addressed to:

Secretary, IEEE-SA Standards Board  
445 Hoes Lane  
Piscataway, NJ 08854-4141  
USA

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; +1 978 750 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 1450.1-2005, IEEE Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450-1999) for Semiconductor Design Environments.

The Standard Test Interface Language (STIL) was initially developed by an ad hoc consortium of automatic test equipment vendors (ATE), electronic design automation vendors (EDA), and integrated circuit (IC) manufacturers to address the lack of a common solution for transferring digital test data from the generation environment to the test equipment.

The scope of the initial STIL standard was limited to satisfy the basic needs of pattern definition. Additional capabilities are developed as separate projects resulting in separate (dot) extensions to the initial STIL standard. The scope of this extension is defined in 1.1 and is primarily to address design needs.

Whereas the initial STIL standard was developed by reviewing many languages already in existence in the industry, this standard has been developed by inventing new capabilities in support of new device designs. The new language constructs have been added such that they do not alter in any way the initial definition of STIL, yet are syntactically compatible with the initial STIL language.

Much of the work to develop and validate these extensions has been done by prototyping on the part of the contributing companies.

## Notice to users

### Errata

Errata, if any, for this and all other standards can be accessed at the following URL: <http://standards.ieee.org/reading/ieee/updates/errata/index.html>. Users are encouraged to check this URL for errata periodically.

### Interpretations

Current interpretations can be accessed at the following URL: <http://standards.ieee.org/reading/ieee/interp/index.html>.

### Patents

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 or patent applications for which a license may be required to implement an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

## Participants

At the time this standard was completed, the 1450.1 Working Group had the following membership:

**Tony Taylor, Chair**  
**Greg Maston, Vice Chair**

Tom Bartenstein  
John Cosley

Daniel Fan  
Bruce Kaufman  
Jose Santiago

Douglas Sprague  
Peter Wohl

The following members of the individual balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Chris Bagge  
Britt Brooks  
Dwayne Burek  
Keith Chow  
Antonio M. Cicu  
Luis Cordova  
John Cosley  
Frans De Jong  
Peter Decher  
Jason Doege  
Dave Dowding  
Geir Eide  
Daniel Fan  
Randall Groves  
William Hanna  
Peter Harrod

Jim Heaton  
Rohit Kapur  
Bruce Kaufman  
James Kemerling  
Adam Ley  
Maurice Lousberg  
Gregory Luri  
Yuhai Ma  
Kevin Marquess  
Denis Martin  
Greg Maston  
Gary Michel  
Yinghua Min  
James Monzel  
Zainalabedin Navabi

Charles Ngethe  
Jim O'Reilly  
Don Organ  
Serafin A. Perez-Lopez  
Vikram Punj  
Mike Ricchetti  
Gordon Robinson  
James Ruggieri  
Jose Santiago  
Gil Shultz  
Douglas Sprague  
Tony Taylor  
Scott Valcourt  
Srinivasa Vemuru  
Gregg Wilder  
Peter Wohl

When the IEEE-SA Standards Board approved this standard on 9 June 2005, it had the following membership:

**Steve M. Mills, Chair**  
**Richard H. Hulett, Vice Chair**  
**Don Wright, Past Chair**  
**Judith Gorman, Secretary**

Mark D. Bowman  
Dennis B. Brophy  
Joseph Bruder  
Richard Cox  
Bob Davis  
Julian Forster\*  
Joanna N. Guenin  
Mark S. Halpin  
Raymond Hapeman

William B. Hopf  
Lowell G. Johnson  
Hermann Koch  
Joseph L. Koepfinger\*  
David J. Law  
Daleep C. Mohla  
Paul Nikolich

T. W. Olsen  
Glenn Parsons  
Ronald C. Petersen  
Gary S. Robinson  
Frank Stone  
Malcolm V. Thaden  
Richard L. Townsend  
Joe D. Watson  
Howard L. Wolfman

\*Member Emeritus

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

Satish K. Aggarwal, *NRC Representative*  
Richard DeBlasio, *DOE Representative*  
Alan Cookson, *NIST Representative*

Michelle Turner  
*IEEE Standards Project Editor*

# Contents

1.	Overview.....	1
	1.1 Scope.....	2
	1.2 Purpose.....	3
2.	Definitions, acronyms, and abbreviations.....	3
	2.1 Definitions .....	3
	2.2 Acronyms and abbreviations .....	4
3.	Structure of this standard .....	4
4.	STIL syntax description .....	5
	4.1 Reserved words .....	5
	4.2 Reserved characters .....	6
	4.3 Reserved UserFunctions .....	7
	4.4 Signal and group name characteristics.....	8
	4.5 STIL name spaces and name resolution .....	8
5.	Expressions .....	9
	5.1 Constant and variable expressions .....	9
	5.2 Expression delimiters—single quotes and parentheses .....	9
	5.3 Arithmetic expressions—integer, real, time, boolean.....	11
	5.4 Pattern data expressions.....	12
	5.5 Expression processing.....	14
	5.6 Boolean—boolean_expr .....	18
	5.7 Integers—integer_expr.....	18
	5.8 Logic expressions—logic_expr .....	19
	5.9 Real expressions—real_expr .....	20
	5.10 Addition to timing expressions—time_expr.....	21
	5.11 SignalVariables—sigvar_expr.....	22
	5.12 Formal parameters in procedures and macros .....	24
	5.13 Integer lists—integer_list.....	24
6.	Statement structure and organization of STIL information .....	25
7.	STIL statement.....	25
	7.1 STIL syntax.....	26
	7.2 STIL example .....	26
8.	UserKeywords statement .....	26
	8.1 UserKeywords syntax .....	26
	8.2 UserKeywords example.....	26

9.	Variables block .....	27
	9.1 Variables block syntax .....	27
	9.2 Variables example.....	29
	9.3 Variables scoping.....	29
	9.4 Variables synchronizing .....	31
10.	Signals block.....	32
	10.1 Signals block syntax .....	33
	10.2 Signals example .....	33
	10.3 Bracketed signal notation enhancement .....	34
11.	SignalGroups block.....	35
	11.1 SignalGroups syntax .....	35
	11.2 SignalGroups, WFCMap, and Variables example.....	35
	11.3 Default WFCMap attribute value .....	36
	11.4 Defining indexed signal groups .....	36
12.	PatternBurst block.....	37
	12.1 PatternBurst syntax .....	37
	12.2 PatternBurst example .....	39
	12.3 Tiling and synchronization of patterns .....	40
	12.4 If and While statements .....	42
13.	Timing block and WaveformTable block .....	43
	13.1 Additional domain specification .....	43
	13.2 CompareSubstitute operation—s, S .....	43
14.	ScanStructures block.....	44
	14.1 ScanStructures syntax .....	44
	14.2 Scan cell naming—cell_ref, chain_ref, cell_group, chain_group .....	47
	14.3 Scoping rules for ScanStructure blocks .....	48
	14.4 Example indexed list of scan cells .....	49
	14.5 Example of ScanChainGroups and ActiveScanChain .....	49
	14.6 Scan chain segments and cell groups.....	51
15.	Pattern data .....	52
	15.1 Data content read back—\C, \D, \E, \S, \U, \W .....	53
	15.2 Vector data mapping and joining—\m, \j .....	55
	15.3 Specifying event data in a pattern—\e.....	57
	15.4 Using expressions within pattern data .....	58
16.	Pattern statements .....	59
	16.1 Additional Pattern syntax.....	59
	16.2 Vector data constraints—F, E.....	61
	16.3 Shift and LoopData statements .....	62
	16.4 Loop statement using an integer expression .....	64
	16.5 MergedScan function.....	65

17.	Procedure and macro data substitution .....	65
	17.1 Nested procedure and macro cells .....	65
	17.2 Passing parameters to variables .....	66
	17.3 Default value of formal parameters .....	67
	17.4 Data substitution using WFCConstant and SignalVariable .....	67
18.	Environment block.....	69
	18.1 Environment syntax .....	69
	18.2 MAP_STRING syntax.....	71
	18.3 NameMaps example .....	71
	18.4 Compact scan-cell mapping using InheritNameMap.....	73
19.	Pragma block .....	74
	19.1 Pragma syntax.....	74
20.	PatternFailReport .....	74
	20.1 PatternFailReport syntax.....	75
	20.2 PatternFailReport example .....	76
	Annex A (informative) Glossary .....	78
	Annex B (informative) Signal mapping using SignalVariables.....	79
	Annex C (informative) Using logic expression with signals.....	83
	Annex D (informative) Using boolean expressions in patterns.....	84
	Annex E (informative) Variables and expressions in algorithmic patterns.....	85
	Annex F (informative) Using AllowInterleave.....	87
	Annex G (informative) Vector data mapping using \m.....	90
	Annex H (informative) Vector data joining using \j .....	93
	Annex I (informative) Block data collection .....	96
	Annex J (informative) Using Fixed and Equivalent statements .....	98
	Annex K (informative) Independent parallel patterns .....	100
	Annex L (informative) Applications using new ScanStructures syntax.....	102
	Annex M (informative) BreakPoints using MergedScan() function.....	106
	Annex N (informative) Labels and X statements for diagnostic feedback.....	109
	Annex O (informative) Use of STIL.1 for specific applications .....	112
	Annex P (informative) Bibliography .....	114



# IEEE Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450<sup>TM</sup>-1999) for Semiconductor Design Environments

## 1. Overview

STIL is an evolving standard being developed in support of various needs for interfacing between test generation tools and test equipment. IEEE Std 1450-1999 (STIL.0) [B3]<sup>1</sup> forms the basis for this evolution. New “dot” standards (like this one) are being developed to address specific needs beyond STIL.0.

This (STIL.1) standard addresses design-related aspects of digital test data. This standard can also be viewed as the addition of advanced features to the STIL.0 baseline to allow for the usage of STIL in more complex applications, while leaving the basic standard unchanged as a vehicle for transmitting basic test data. The following is a brief overview of the new features in STIL.1 to support advanced applications such as (1) embedded cores,<sup>2</sup> (2) families of test patterns, (3) mapping to automated test equipment (ATE) systems,<sup>3</sup> (4) mapping to simulation, and (5) devices with advanced design for test (DFT). Please see Annex O for a list of specific statements for each of these features.

*Environment mapping:* Data for a device exist in many forms and in many other software environments. Examples include (1) simulation environment, (2) static analysis environment, (3) specific ATE system environment. The STIL Environment block is a new mechanism to relate STIL data to these other environments. No assumptions, expectations, or limitations are imposed on the other environments. It is just a way of relating one to the other.

*Parameterized data:* Much of STIL data are declarative in nature (i.e., it defines various static attributes of a device or pattern set). The addition of constant declarations, IntegerConstant and WFCConstant, allows a data set to be created that applies to a family of devices.

*Complex test protocol definition:* Test protocol definitions are usually contained in STIL procedures or MacroDefs and are used to specify the application of a series of data values to a device. STIL.0 supports scan chain data passing and simple WaveformCharacter (WFC) data passing via the # and % characters. STIL.1 enhances this capability by allowing the use of data substitution from SignalVariables and integer

---

<sup>1</sup>The numbers in brackets correspond to those of the bibliography in Annex P.

<sup>2</sup>This standard contains syntax in support of embedded cores. See IEEE Std 1450.6<sup>TM</sup>-2005 (Core Test Language) [B5] for the complete specification.

<sup>3</sup>This standard contains syntax in support of ATE systems. See IEEE P1450.3<sup>TM</sup> (Test Resource Constraints) [B4] for the complete specification.

*Complex test protocol definition:* Test protocol definitions are usually contained in STIL procedures or MacroDefs and are used to specify the application of a series of data values to a device. STIL.0 supports scan chain data passing and simple WaveformCharacter (WFC) data passing via the # and % characters. STIL.1 enhances this capability by allowing the use of data substitution from SignalVariables and integer-expressions. STIL.1 also enhances the functionality of Loops and Vectors and adds If/While decisions on pattern statements. These capabilities are needed for BIST, embedded cores, and various test access mechanisms.

*Advanced scan architecture:* Advanced DFT techniques require additional capabilities beyond what is defined in STIL.0, which includes multistate scan cells, reconfigurable scan-chains, and scan-chain indexing.

*Run-time pattern decisions:* The If, Else, While, and LoopData are new STIL.1 constructs that have been added for specification of pattern activity. These statements are needed in the specification of patterns to be run in the simulation environment. Although there is no standardization among ATE systems on run-time instructions for pattern execution, it is anticipated that restricted versions of these statements will be incorporated into ATE test patterns.

*Pattern burst options:* New variations on the PatternBurst have been added to allow for patterns running in parallel, patterns running in LockStep, and patterns that can be reordered. For parallel pattern execution, the specification for how the patterns fit together can be specified with the Fixed and Extend statements.

*Enhanced user extensibility:* The UserKeyword extensibility defined in STIL.0 has been extended to allow keywords to be defined on a per-block-type basis.

*Signal relationships:* Additional syntax is provided to allow the specification of relationships between signals. This process is preformed via \m to map WFCs to another WFC, \j to join WFCs, Extend to define behavior of signals beyond the bounds of a given pattern, and Fixed to restrict any further changes to signals within a pattern.

*Fail feedback:* Three new features are added to facilitate the processing of failure data from an ATE system back to design tools. The first is the X or cross-reference statement that allows the specification of where in a pattern/vector sequence a failure occurs. The second is the FailFeedback block for reporting fails. The third is the S/s timing event that allows for the specification of a data capture protocol for the purpose of capturing bulk fail data for processing.

## 1.1 Scope

Structures are defined in STIL to support usage as semiconductor simulation stimulus, including (1) mapping signal names to equivalent design references, (2) interface between scan and built-in self test (BIST) and the logic simulation, (3) data types to represent unresolved states in a pattern, (4) parallel or asynchronous pattern execution on different design blocks, and (5) expression-based conditional execution of pattern constructs.

Structures are defined in STIL to support the definition of test patterns for sub-blocks of a design<sup>4</sup> (i.e., embedded cores) such that these tests can be incorporated into a complete higher level device test.

Structures are defined in STIL to relate fail information from device testing environments back to original stimulus and design data elements.

---

<sup>4</sup>Syntax in this document that is used in the definition of patterns for sub-blocks is summarized in Annex O.

## 1.2 Purpose

The STIL language definition is enhanced to support the usage of STIL in the design environment, which includes extending the execution concept to support STIL as a stimulus language, to allow STIL to be used as an intermediate form of data, and to allow STIL to capture design information needed to port simulation data to device test environments.

In addition, define extensions to support the definition of subelement tests and to define the mechanisms to integrate those tests into a complete device test. This effort is to be performed in conjunction with IEEE Std 1500™-2005 [B6] and IEEE P1450.6 [B5], which are defining standards for the definition and integration of embedded cores.

Finally, define the constructs necessary to correlate test failure information back to the design environment, to allow debug and diagnosis operations to be performed based on failure information in STIL format.

## 2. Definitions, acronyms, and abbreviations

### 2.1 Definitions

For the purposes of this standard, the following terms and definitions apply. Additional terminology specific to this standard is found in Annex A. *The Authoritative Dictionary of IEEE Standards Terms* [B1] should be referenced for terms not defined in this clause.

**2.1.1 automated test equipment (ATE):** It refers to a tester that is capable of interfacing to a semiconductor device and executing test pattern data that is imported from a STIL file/stream.

**2.1.2 built-in self-test (BIST):** A design practice in which test logic is incorporated into the circuitry of a semiconductor device. This circuitry may provide completely autonomous testing of a device (i.e., without any requirement of a tester). It may be such that stimulation by an external tester is required; however, the STIL file may be substantially different from a device without this circuitry incorporated.

**2.1.3 core:** A component or module that contains separately developed functionality, integrated into a chip to provide additional overall functionality. *See also: System on Chip.*

**2.1.4 electronic design automation (EDA):** The set of software tools that are used for the design and creation of semiconductor chips. It includes the software tools that create the test patterns for the chips, which are often referred to as automated test pattern generators (ATPGs).

**2.1.5 Standard Test Interface Language (STIL):** The set of IEEE standards, including IEEE Std 1450-1999, and all dotted extensions, including this one.

**2.1.6 System on Chip (SoC):** An integrated circuit containing modules that are designed/integrated such that they can be tested independently and have associated test patterns for each module.

**2.1.7 WaveformCharacter (WFC):** A symbol used for referencing waveforms.

NOTE—See Annex A.<sup>5</sup>

<sup>5</sup>Notes in text, tables, and figures are given for information only, and do not contain requirements needed to implement the standard.

**2.1.8 WaveformTable (WFT):** An STIL block statement used to define waveforms across multiple signals and WFCs.

NOTE—See Annex A.

**2.1.9 WaveformGenerationLanguage (WGL):** A proprietary standard that was, in part, used as the basis for STIL.0.

## 2.2 Acronyms and abbreviations

ATE	automated test equipment
BIST	built-in self-test
DFT	design for test
DUT	device under test
EDA	electronic design automation
STIL	Standard Test Interface Language
STIL.0	IEEE Std 1450-1999 [B3]
STIL.1	this standard
WFC	WaveformCharacter
WFT	WaveformTable
WGL	WaveformGenerationLanguage

## 3. Structure of this standard

This standard is an adjunct to STIL.0. The conventions established and defined in STIL.0 are used in this standard and are included verbatim.

Many clauses in this standard add additional constructs to existing clauses in STIL.0 and are so identified in the title. The Environment block is a new construct introduced in this standard. All clauses in this standard are normative. Example code is provided within each clause. More complete examples are provided in the annexes, which are informative.

The following is a copy of the conventions as defined in STIL.0 and adhered to in this standard.

Different fonts are used as follows:

- SMALL CAP TEXT indicates user data.
- Courier text indicates code examples.

In the syntax definitions

- a) SMALL CAP TEXT indicates user data.
- b) **Bold text** indicates keywords.
- c) *Italic text* references metatypes.
- d) () indicates optional syntax that may be used zero or one time.
- e) ()+ indicates syntax that may be used one or more times.
- f) ()\* indicates optional syntax that may be used zero or more times.
- g) <> indicates multiple-choice arguments or syntax.

In the syntax explanations, the verb “shall” indicates mandatory requirements. The meaning of a mandatory requirement varies for different readers of the standard:

- To developers of tools that process STIL (readers), “shall” denotes a requirement that the standard imposes. The resulting implementation is required to enforce this requirement and issue an error if the requirement is not met by the input.
- To developers of STIL (writers), “shall” denotes mandatory characteristics of the language. The resulting output must conform to these characteristics.
- To the users of STIL, “shall” denotes mandatory characteristics of the language. Users may depend on these characteristics for interpretation of the STIL source.

The language definition clauses contain statements that use the phrase “it is an error” and “it may be ambiguous.” These phrases indicate improperly defined STIL information. The interpretation of these phrases will differ for the different readers of this standard in the same way that “shall” differs, as identified here in the dashed list.

## 4. STIL syntax description

This clause defines extensions to STIL.0, Clause 6.

All constructs and restrictions for STIL.0, Clause 6 are in effect here, with the following additions:

- Additional STIL reserved words are defined (see Table 1) for the top-level blocks specified within this standard.
- Additional STIL reserved characters are defined (see Table 2) for the new characters specified within this standard.
- Additional definition of signal and group naming, as well as name space resolution, is provided in this clause.
- Extensions to the expression environment are defined in this standard. Expression definitions (which are part of Clause 5 in STIL.0) are now specified in a separate clause (Clause 6 in this standard).

### 4.1 Reserved words

Table 1 lists all STIL reserved words defined by this standard. Only top-level block names that are defined in this standard are added to the STIL reserved word list. New keywords that appear inside of top-level blocks are not restricted from usage in other contexts outside of the definition of that keyword. No change to reserved words as defined in STIL.0 is made by this standard.

**Table 1—Additions to STIL reserved words**

Environment
PatternFailReport, Pragma
Variables

## 4.2 Reserved characters

Several reserved characters identified in STIL.0 are applied in additional contexts for this standard. Table 2 lists additional STIL reserved characters defined in this standard as well as additional contexts of previously identified reserved characters. No change to reserved characters as defined in STIL.0 is made by this standard. New reserved characters are so indicated in column two. All other characters in this table have extended capability beyond that defined in STIL.0.

**Table 2—Additions to STIL reserved characters**

Char	New in STIL.1	Usage
!		<b>exclamation (NOT sign):</b> An inversion operator in expressions
%		<b>percent sign:</b> Is used as the modulus in expressions and for parameter passing to procedures and macros
()		<b>parentheses:</b> Are used in expressions
*		<b>multiply:</b> Is used in expressions
+		<b>add:</b> Is used in expressions
-		<b>subtract:</b> Is used in expressions
&	YES	<b>and:</b> Is used in expressions
	YES	<b>or:</b> Is used in expressions
^	YES	<b>xor:</b> Is used in expressions
~	YES	<b>bit-wise negation:</b> Is used in logic expressions <b>error return value:</b> Is used when \W has no WFC chars to return
<		<b>less than:</b> Is used in integer and real expressions
>		<b>greater than:</b> Is used in integer and real expressions
<=		<b>less than or equal to:</b> Is used in integer and real expressions
>=		<b>greater than or equal to:</b> Is used in integer and real expressions
==		<b>equal to:</b> Is used with pattern data expressions (i.e., to compare strings of WFCs)
!=		<b>not equal to:</b> Is used with pattern data expressions (i.e., to compare strings of WFCs)

**Table 2—Additions to STIL reserved characters (continued)**

Char	New in STIL.1	Usage
:==	YES	<b>equal to:</b> Is used with integer and real expressions (i.e., to compare integer or real numbers)
<>	YES	<b>not equal to:</b> Is used with integer and real expressions (i.e., to compare integer or real numbers)
?:		<b>conditional:</b> Is used in expressions
=		<b>assignment:</b> Is used in signal variable expressions, vector data, group name definitions, spec category expressions, and spec variable expressions
:=	YES	<b>assignment:</b> Is used in integer and real expressions
_	YES	<b>underscore:</b> Is used as a separator in integer values
.		<b>dot:</b> Is used to designate macro and procedure calls in FailData block
:		<b>colon:</b> Is used to designate subcomponents of a design (i.e., embedded cores)
::	YES	<b>double colon:</b> Is used to designate a domain reference in expressions
{* *}		<b>brace, asterisk:</b> Is used to designate a block of data in the case of Pragma (same as delimiters used for annotation blocks)

### 4.3 Reserved UserFunctions

The user function names in Table 3 are reserved for use in expressions.

**Table 3—UserFunctions**

Name	New in STIL.1	Usage
min		<b>function:</b> Is used in integer or real expressions to return the minimum value from a list of values (as defined in STIL.0).
max		<b>function:</b> Is used in integer or real expressions to return the maximum value from a list of values (as defined in STIL.0).
MergedScan	YES	<b>function:</b> Is used in patterns to test whether scan in/out data have been merged. See 16.5.

#### 4.4 Signal and group name characteristics

This clause defines extensions to STIL.0, Subclause 6.10.

All constructs and definitions in STIL.0, Subclause 6.10 are in effect. In addition, signals expressed with a bracketed construct shall allow the use of a previously declared integer-constant, in which integer values are allowed. See 5.1 for the definition of IntegerConstant.

#### 4.5 STIL name spaces and name resolution

Additions to STIL.0, Subclause 6.16.

The Environment block augments the STIL name space as defined in Table 4. This table is incremental to STIL.0, Table 6; all definitions present in that table remain unchanged.

Variable names and constant names are used by expressions. This name space also contains signal names, signal group names, and spec variables defined for the context of that expression. Referenced variable-names shall be unique against all signal names, signal groups names, and spec variable names defined for a PatternBurst or application of a Pattern referenced by that PatternBurst.

**Table 4—Additions to STIL name spaces**

STIL block	Type of name	Domain restrictions
Environment	Environment domain names	Supports a single unnamed block and domain named blocks. Domain names shall be unique across all Environment blocks.
Variables	Variables domain names	Supports a single unnamed block and domain named blocks. Domain names shall be unique across all Variables blocks.
Variables, Signals, SignalGroups, Spec	Signal names SignalGroup names SignalVariable names Spec variable names Integer names IntegerConstant names WFCConstant names	Names present in this name space are dependent on the domain reference statements in the PatternBurst (SignalGroups domains, Variables domains) and the PatternExec (Category domains). It is an error for defined Variable or Constant names to conflict with Signals, SignalGroups, or Spec variable names accessible in the same context.
ScanStructures	ScanCell names ScanChain names (also used for scan segments and cell groups)	Names present in this space are dependent on the domain reference statements for ScanStructures in the PatternBurst. Scan cell names, scan chain names, and cell group names in this common name space are unique except if an unnamed ScanStructures block exists. Names present in an unnamed block are available unless hidden by a definition of the same name in a named and referenced ScanStructure block.

## 5. Expressions

This clause defines extensions to STIL.0, Clause 6.

STIL.0 defines a limited usage of expressions; see STIL.0, Subclause 6.13 and Subclause 6.14. This standard extends these expression capabilities with additional variable types (SignalVariable, Integer, IntegerConstant, WFCConstant) and additional expression constructs. The use of operators and the expression constructs as defined for STIL.0 is unchanged. The full set of STIL.0 and STIL.1 expression operations are defined in Table 7. The detail of syntax and usage of the new expressions are defined in 5.1 – 5.13.

### 5.1 Constant and variable expressions

Expression constructs (which are defined in 5.2 – 5.13) can be divided into two classes: those that can be completely resolved at parse time (i.e., constant expressions) and those that can only be resolved at run time (i.e., variable expressions).

Constant expressions are those that contain only literal values (e.g., ‘5ns’ or ‘10ns’), or named constants (e.g., Spec-Variable, IntegerConstant, and WFCConstant). As in STIL.0, statements that contain constant expressions can be fully resolved once the STIL file/stream has been fully parsed and the domain references have been resolved. Constant expressions can be used anywhere that the syntax definitions allow a literal value. The use of constant expressions provides for (1) improved readability, (2) parameterized STIL data, (3) enhanced reuse, and (4) improved maintainability.

Variable expressions are those that contain named variables, such as SignalVariable and Integer. The use of variable expressions is intended primarily to support “design” applications. The ability of an ATE system to support variable expressions may be limited, if possible at all.

Both constant and variable expressions can be further classified into “arithmetic expressions” and “pattern data expressions.” The allowed constructs of each of these expression types are defined in 5.2 – 5.13.

### 5.2 Expression delimiters—single quotes and parentheses

Single quotes are defined in STIL.0 as the delimiter to be used around expressions. This usage is unchanged with the introduction of new expression capabilities in STIL.1, and a STIL.0 file is completely compatible with all STIL.1 rules. However, the rules governing the use of single quotes and parentheses are relaxed. The following rules apply to the use of single quotes in STIL.1:

- a) Single quotes, when used, are always the outermost symbols of an expression.

```
'T2+5ns' // timing expression
'SIG1+SIG2+SIG3' // signal group expression
```

- b) Assignment and boolean operators shall not be used inside a single-quoted expression.

```
V { 'INT := INT+5' ; } // illegal
If 'INT == 5' { } // illegal
```

- c) Single quotes shall not be nested.

```
V { 'BUS [1..K+1]' = XXX ; } // illegal - nesting of quotes
```

- d) Wherever expressions are allowed, a single token may be used without delimiters. A single token may be a literal value, a named constant, or a named variable.

```
V { INT := '5' ; } // single token with quotes
V { INT := 5 ; } // single token without quotes
V { INT1 := INT2 ; } // single token without quotes
```

- e) Wherever expressions are allowed, they may be represented without delimiters, with the exception that if a multiterm expression is within a complex data stream, then the delimiters are required. Examples of cases in which delimiters are required are shown in the later examples, using parentheses.

```
V { INT := 'INT+1' ; }
V { INT := INT+1 ; }
```

- f) Parentheses may be used inside a single-quoted expression.

```
V { INT := '(INT+1)*2' ; }
V { '(GRP-SIG)+SIG' = 00001 ; } // make SIG the last WFC in the list
```

Parentheses are the preferred method of delimiting expressions in STIL.1. Parentheses can be used around the entire expression or inside the expression to specify the order of processing. The following rules apply to the use of parentheses:

- g) Parentheses are allowed anywhere that single quotes are allowed in STIL.0.

```
V { (SIG1+SIG2) = XX ; }
01 { (T2+5ns) D/U ; }
```

- h) Assignment expressions may be constructed with or without the use of delimiters.

```
V { INT := INT+1 ; }
V { INT := (INT+1) ; }
V { (INT := INT+1) ; }
V { INT := (X+2)*2 ; } // parens required to define order
V { SIG1+SIG2 = XX ; }
V { (SIG1+SIG2) = XX ; }
V { (SIG1+SIG2 = XX) ; }
```

- i) Parentheses are not required to delimit boolean expressions; however, in practice, they are commonly used.

```
If (\W(A+B) == \wxx) { }
If \W(A+B) == \wxx { }
If (\WGRP != \w\r(K+12) X) { }
If \WGRP != \w\r(K+12) X { }
```

- j) Parentheses may be nested.

```
V {SIG = \r(K+12) X ; }
If (\WBUS[1..(K-1) != \wXXX) { }
```

- k) Parentheses are used to delimit parameters to functions within expressions.

```
V { INT := Min(X, Y) ; }
```

- l) If a multiterm expression is within a complex data stream, then delimiters around the expression are required.

```
SIG1 + SIG2 = XX;
\W(SIG1 + SIG2) == XX; // parens required
AB { 2ns D/U ; }
AB { TIM D/U ; }
AB { (TIM + 2ns) D/U ; } // parens required
V { GRP = \r2 X YZ ; }
V { GRP = \rTWO X YZ ; }
V { GRP = \r(TWO+1) X YZ ; } // parens required
V { GRP[5 9] = XX ; }
V { GRP[K5 K9] = XX ; }
V { GRP[(KK+5) (KK+9)] = XX ; } // parens required
```

### 5.3 Arithmetic expressions—integer, real, time, boolean

Arithmetic expressions are used in various STIL statements. In the statement syntax definition within this standard, the reference identifiers, *integer\_expr*, *real\_expr*, *boolean\_expr*, and *time\_expr*, indicate that an arithmetic expression is expected, and they signify the expected output from the expression evaluation. It is allowed to use an arithmetic expression anywhere a literal arithmetic value is allowed as long as it contains only constants and literals. This subclause defines the common rules for all arithmetic expressions.

Arithmetic expressions use the following operators:

- := assignment operator
- ?: conditional operator
- := compare for equality; return integer 1 if equal; else return 0
- <> compare for inequality; return integer 1 if not equal; else return 0
- < compare less than; return integer 1 if less; else return 0
- > compare greater than; return integer 1 if greater; else return 0
- <= compare less than or equal to; return integer 1 if less or equal; else return 0
- >= compare greater than or equal to; return integer 1 if greater or equal; else return 0

The one exception to the use of := for integer assignments is in the Spec block. For compatibility with STIL.0, it is allowed to use either a bare '=' or the ':=' in this context.

Arithmetic expressions use the following operands:

- Integer, IntegerConstant that are in scope according to the Variables context
- Spec variables that are in scope according to the Category selection
- Timing event labels when used inside a Timing block as defined in STIL.0
- Literal integers, real numbers, and engineering numbers

Pairs of single quotes are used as delimiters in arithmetic expressions according to the rules as defined in STIL.0. It is not allowed to nest single quotes, and the usage should be limited to simple expressions (e.g., 'T2+5ns'). Some statements require that delimiters always be used around expressions, even when the expression is a single literal value or name. If not so stated in the statement definition, then single operands may be used without any delimiters.

Paired parentheses are the preferred method of delimiting expressions in STIL.1. Parentheses can be used around the entire expression or inside the expression to specify the order of execution.

Multiple assignment operators are not allowed in an arithmetic expression:

```
V { INT1 := INT2 := 0; } //illegal use of multiple assignment operators
```

The following examples are of keyword identified expressions:

```
Loop 50 { }
ScanLength 100;
Loop K+5 { }
Loop 'K+5' { }
Loop (K+5) { }
```

The following examples are of assignment expressions:

```

Condition { INT := 5; }
V { INT := 5; }
C { INT := 5; }
C { INT := INT + 1; }
C { INT := K; }
C { INT := (INT==0) ? 99 : INT-1; }

```

The following examples are of boolean expressions:

```

If (INT == 6) { }
If (INT >= K*2) { }
If (INT > SPEC) { }
If (SPEC < 250ns) { }
While (INT <> 13) { }

```

The following example is of an arithmetic expression within a pattern data expression:

```
V { GRP[1..(K+1)] = \r(K+1) X; }
```

## 5.4 Pattern data expressions

Whereas arithmetic expressions operate on numeric data, pattern data expressions operate on lists of WFCs. To differentiate the two, a different set of operators is used:

- = assignment operator
- == compare for equality; return integer 1 if equal; else return 0
- != compare for inequality; return integer 1 if not equal; else return 0
- ?: conditional operator
- The pattern data operators that are preceded with a backslash (see Clause 15)

Pattern data expressions use the following operands:

- Signals
- SignalGroups that are in scope according to the PatternBurst context
- Literal lists of WFC's
- SignalVariables and WFCConstants that are in scope according to the Variables context
- Formal parameters from Procedures and Macros

Single quotes are used as delimiters in pattern data expressions according to the rules as defined in STIL.0. It is not allowed to nest single quotes, and the usage should be limited to identifying signal groups (e.g., 'SIGA+SIGB,' 'BUS[1..10]'). Some statements require that delimiters always be used around expressions, even when the expression is a single literal value or name. If not so stated in the statement definition, then single operands may be used without any delimiters.

Unlike arithmetic expressions, parentheses are not to be used on pattern data assignment expression. Assignments should be of the form: TWOSIGS=XX; or 'SIGA+SIGB'=XX;. Parentheses may be used when needed to control order of evaluation as in the following: '(XBUS-XBUS[5])+YBUS'=11110000;.

Multiple assignment operators are not allowed in pattern data expressions:

```
V { SIG1 = SIG2 = X; } // illegal - use of multiple assignment operators
```

The following examples are of pattern data assignment expressions.

```
V { SIG = X; } // assign a WFC to a signal
V { GRP = XXXXXX; } // assign WFCs to a signal group
V { SIGVAR = XXX; } // assign WFCs to a signal variable
V { GRP[5..6] = XX; } // selective assign of WFCs
V { 'A+B+C' = XXX; } // assign WFCs to multiple signals
V { SIGVAR = \W SIG; } // assign WFC from a signal to a variable
V { SIG = \W SIGVAR[3]; } // assign WFC from a variable to a signal
V { GRP = \W HALT; } // assign a WFC-const to a signal group
V { GRP[1..12] = \W SIGVAR[1..4] XXXX \W SIGVAR[5..8]; }
```

The following examples are of pattern data boolean expressions. Note the use of \W on both sides of the expression. The \W operators is required to indicate that WFC data are being compared. The \w operator is used to extract WFC data from a named entity. The \w operator is used to indicate that literal WFC data follow. Note also that when literal WFC data are used, it may appear on either side of the expression.

```
If (\WSIGVAR[6..5] == \wXX) { }
If (\wXX == \WSIGVAR[6..5]) { }
If (\WSIGVAR1 != \WSIGVAR2) { }
```

The following example is of a pattern data expression inside an arithmetic expression:

```
V { INT := (\WGRP==\w010) ? 1 : 0; }
```

The following complete example of statements contains both arithmetic and pattern data expressions:

```
1: STIL 1.0 { Design 2005; }
2: Header {
3:     Source "STD 1450.1-2005";
4:     Ann {* sub-clause 5.4 *}
5: }
6: Signals { S[1..4] In; }
7: SignalGroups { SIGGRP = 'S[1..4]'; }
8: Variables {
9:     IntegerConstant RUN := 0;
10:    IntegerConstant EXIT := 1;
11:    IntegerConstant LOAD := 2;
12:    IntegerConstant UNLOAD := 3;
13:    Integer CMD; //use values RUN, EXIT, LOAD, UNLOAD
14:    Integer INT;
15:    Integer HEX {InitialValue 0x1000;}
16:    Integer CC;
17:    Integer DD;
18:    SignalVariable SIGVAR1[1..4];
19:    WFCConstant STOP=00;
20:    WFCConstant GO=01;
21:    WFCConstant RESET=10;
22:    SignalVariable SIGVAR2[1..2]; //use values STOP, GO, RESET
23: }
24: Spec {
25:     Category CAT {
26:         TIME = '25ns';
27:     }
28: }
29: Pattern PAT {
30:     C {INT := 1;}
```

```
31:      If (INT) {} Else {}           // Execute the If block, INT is True
32:      C { INT := 0; }
33:      If (INT) {} Else {}           // Execute the Else block, INT is False
34:      If (INT == 956) {}            // a boolean expression
35:      If (TIME >= 20ns) {}          // a real number in engr units
36:      If (CC <= DD) {}              // test for CC less than or equal to DD
37:      If (INT < min(CC,DD)) {}      // use of the min function
38:      C {INT := 5;}                 // set variable INT equal to 5
39:      C {CMD = \W RUN;}             // set using constant definition
40:      C {HEX := 0xFF;}              // set integer to 255
41:      If (CMD <> LOAD) {}           // integer compare to a constant
42:      C {INT := INT+1;}             // integer expression
43:      C {SIGVAR1 = LH01;}           // set signal variable to the string LH01
44:      V {SIGGRP = \WSIGVAR1;}      // set signals equal to a variable
45:      If (\WSIGVAR2 == \WSTOP) {}   // compare with a constant
46:      If (\WSIGVAR2 == HH) {}       // compare with a WFC list
47: } // end Pattern
```

## 5.5 Expression processing

This subclause defines the rules for evaluating expressions. Please refer to 5.6 – 5.12 for the definition of the expression types.

Expressions are built up in a hierarchical manner; internal subexpressions are processed, and they, in turn, are used in the next level of the hierarchy. A subexpression is legal expression syntax that is used within another expression. The only distinction between an expression and a subexpression is that a subexpression is a part of another expression.

At the lowest level are the primitive terms (Table 5). A primitive term used by itself represents a legal expression. Each primitive term represents a value and a type. In some cases, the value may not be determinable until run time. In all cases, the type is determinable when the STIL file/stream is parsed.

Subexpressions may be combined with operators to form subexpressions at the next higher level in the expression hierarchy. Like the primitive terms, each subexpression represents a value and a type. And, as with the primitive terms, the value may not be determinable until run time, but the type is determinable when the STIL file/stream is parsed.

**Table 5—Primitive terms**

Term	Type	Example—Comment
Integer	numeric	5, 1066
Hex integer	numeric	0xFFFF, 0x123
SI units	numeric	2mV, 25ns —SI units are defined in Table 3 and Table 4 of STIL.0
Real	numeric	2e+3, 4.5e6, 25e-9
Time event reference	numeric	@, @1, @@, @T1, @T1.1 —used in the context of a Timing->Waveform block
Time label	numeric	TXX: —used in the context of a Timing->Waveform block
Signal Signal group	sigref	NAME, "NAME"
Signal Signal group Signal variable WFC constant	WFC list	\WNAME, \W NAME, \W "NAME" —use \W to extract the WFC list from the named entity
String	WFC list	10HLXT, \w 10HLXT, \h01 FF, \d01 255 —the \ is required with the boolean operators (== <>) —the dot notation for concatenation is neither required or allowed in WFC data; a whitespace character shall serve as separator for WFC lists of greater than 1024 characters
Integer list	integer list	1359 —an ordered set of integers, typically used in the formation of signal groups and scan chains
Bare string	string	STRING, "STRING", "STRING WITH SPACES", "STRING WITH SPECIAL CHARS @\$%^" —used to reference named entities
Concatenated string	string	XXX.YYY yields XXXYY —used when a string is greater than 1024 characters
Domain reference	string	DOM::NAME —used to reference an item within a named block, i.e., within a named SignalGroups, Timing, or ScanStructures block
Subcomponent reference	string	COMP:NAME, COMP:DOM::NAME —reserved for referencing subcomponents of a design (i.e., embedded cores)
	boolean	—a boolean type is the result of a boolean operation, i.e., ==, :=, ! (see Table 7) —integer 0 is returned for a boolean true result, and integer 1 for a boolean false result —the values TRUE and FALSE are not defined —there is no variable type for boolean

Table 6 defines implicit casting rules that are allowed on an expression to change a number from one type of primitive term to another. The casting rule shall be used when necessary to (1) conform to the requirements of the next level of hierarchy, (2) conform to the requirements of the statement containing the expression, or (3) compare a real to an integer number.

**Table 6—Implicit numeric casting rules<sup>a</sup>**

Term	Cast to	Comment
SI units	real	—used when expression requires a real number
real	integer	—used when expression requires an integer value —the fractional part is truncated; i.e., 3.7 yields 3
integer	real	—used when comparing an integer to a real number
integer	boolean	—used when expression requires a boolean value —value of 1 or greater yields TRUE —value of 0 or negative yields FALSE

<sup>a</sup>If the required behavior is different from the implicit type casting rules, then explicit casting should be performed via user functions.

Table 7 is expanded from Table 5 of STIL.0 and includes the new STIL.1 operators. The behavior of the logic operators is defined in IEEE Std 1364™-2001 (Verilog) [B2]. This table is ordered by precedence. Operators within each double-line separated group are of equal precedence and are processed from left to right. Each double-line separated group of operators has higher precedence than the groups below them and shall be processed first.

**Table 7—Operators and functions allowed in expressions**

Operator	New	Definition	Number operands	Requirements of operands	Type of the result
Min ( )		minimum value function	>=1	all of same numeric type	same as operands
Max ( )		maximum value function	>=1	all of same numeric type	same as operands
( )		parentheses	1	any	same as operand
[ ]		square brackets	1	integer or integer list	integer list
,		comma - used as an argument separator in functions	2	any	same as operands
!	STIL.1	negation (unary)	1	boolean or sigref	same as operand
~	STIL.1	bit-wise negation (unary)	1	integer	same as operand
+		plus (unary)	1	numeric	same as operand
-		minus (unary)	1	numeric	same as operand
/		divide	2	numeric	numeric, SI units adjusted per rules

**Table 7—Operators and functions allowed in expressions (continued)**

Operator	New	Definition	Number operands	Requirements of operands	Type of the result
*		multiply	2	numeric	numeric, SI units adjusted per rules
%	STIL.1	modulus	2	integer	integer
+		add	2	same numeric type	same as operands
-		subtract	2	same numeric type	same as operand
+		concatenate signal list	2	signal or signal group	signal group
-		remove from signal list	2	signal or signal group	signal group
..		integer range (two consecutive dots)	2	integer	integer list
	STIL.1	whitespace	2	integer or integer list	integer list
<	STIL.1	less than	2	same numeric type	boolean
>	STIL.1	greater than	2	same numeric type	boolean
<=	STIL.1	less or equal	2	same numeric type	boolean
>=	STIL.1	greater or equal	2	same numeric type	boolean
==	STIL.1	equal (WFC)	2	WFC list	boolean
!=	STIL.1	not equal (WFC)	2	WFC list	boolean
:=	STIL.1	equal (numeric)	2	same numeric type	boolean
<>	STIL.1	not equal (numeric)	2	same numeric type	boolean
&	STIL.1	bit-wise and	2	integer or sigref	same as operands
^	STIL.1	bit-wise exclusive or	2	integer or sigref	same as operands
^~, ^^	STIL.1	bit-wise equivalence	2	integer or sigref	same as operands
	STIL.1	bit-wise inclusive or	2	integer or sigref	same as operands
&&	STIL.1	and	2	boolean	boolean
	STIL.1	or	2	boolean	boolean
?:		conditional expression	3 (1?2:3)	1: boolean 2,3: same type— numeric or WFC list	same as operands 2, 3
=		assignment (WFC)	2	—left: must be a name of a symbol that can be assigned WFCs —right: a WFC list	WFC list
:=	STIL.1	assignment (numeric)	2	—left: must be a name of a symbol that can be assigned to same type —right: any numeric type	same as second operand

## 5.6 Boolean—*boolean\_expr*

An expression is interpreted as a boolean if the context in which it is used requires a boolean result. A boolean result can be achieved by means of the boolean operators. For pattern data expressions, the boolean operator shall be either `==` or `!=`, and both sides shall be of type WFC (i.e., a list of WFCs preceded by `\w`, else a signal, signal group, signal variable, or WFC-constant preceded by `\W`). For arithmetic expressions, the boolean operator shall be either `:=`, `<`, `<=`, or `>=`, and both sides shall be of type integer or real.

An expression that results in an integer or real value shall be interpreted as a boolean if the usage context so requires; in which case, a nonzero positive value is interpreted as true, whereas a value of zero or negative is interpreted as false. Expressions that evaluate to WFCs shall not be used as booleans.

Although it may be good practice to use parentheses around boolean expressions, this is not a requirement of the language.

The following examples are of boolean expressions:

```

48: STIL 1.0 { Design 2005; }
49: Header {
50:     Source "STD 1450.1-2005";
51:     Ann { * sub-clause 5.6 * }
52: }
53:
54: Variables {
55:     Integer INT;
56:     Integer FLAG;
57:     IntegerConstant FALSE := 0; // FLAG value
58:     IntegerConstant TRUE := 1; // FLAG value
59:     SignalVariable OP_CODE[3..0];
60:     WFCConstant RUN = 1011; // OP_CODE value
61:     WFCConstant STOP = 0111; // OP_CODE value
62:     WFCConstant RESET = 0000; // OP_CODE value
63: } // end Variables
64:
65: Pattern MY_PAT {
66:     If (INT := 13) {}
67:     If (FLAG := TRUE) {}
68:     If (\WOP_CODE == \WRESET) {}
69:     If (\w1111 == \WOP_CODE) {}
70: } // end Pattern

```

## 5.7 Integers—*integer\_expr*

An integer expression is an expression that evaluates to integer. Integer expressions are allowed anywhere that the statement syntax calls for an “*integer\_expr*”. See the list of allowed operators in Table 7. The following rules of interpretation apply to integers and integer expressions:

- A bare integer may be declared either with or without single quotes, e.g., 5 or ‘5’.
- The underscore character may be used as a separator within an integer declaration, e.g., 65\_535.
- If an integer result is called for and the expression results in a number with a fractional part, the fraction is truncated to produce an integer value; e.g., the values [6.1, 6.8, -3.1, -3.8] become [6, 6, -3, -3], respectively. The truncation is performed at the conclusion of the evaluation; i.e.,  $(3/2) \times 2$  results in a value of 3.
- The following formats are not allowed for integers: 56E3, 56K.

- e) Integer may be expressed in hexadecimal by preceding the number with 0x, e.g., 0xFF.

Integer expressions may comprise integer literals, integer variables, integer constants, or multiterm expressions that result in an integer value. The following examples are of integer expressions and their usage:

```

71: STIL 1.0 { Design 2005; }
72: Header {
73:     Source "STD 1450.1-2005";
74:     Ann { * sub-clause 5.7 * }
75: }
76:
77: Signals { SIGA In; SIGB Out; SIGC InOut; FOO[0x0..0x3F] In; }
78: Variables {
79:     Integer II;
80:     Integer JJ { InitialValue 56*1024; }
81:     IntegerConstant KK := 2;
82: }
83: SignalGroups {
84:     ALL = 'SIGA+SIGB+SIGC+FOO';
85: } //end SignalGroups
86:
87: Timing { WaveformTable WFT {
88:     Waveforms {
89:         ALL {
90:             01 { '2ns+KK*0.5ns' U/D; }
91:         }
92:     } //end Waveforms
93: }} //end Timing
94:
95: Pattern P {
96:     C { II := 1234; }
97:     C { II := 48000000; }
98:     C { II := 48_000_000; } // equivalent to 48000000
99:     C { II := II + 2; } // expression with a variable
100:    C { II := 13; JJ := 13; } // expression containing assignment
101:    C { II := JJ/2; } // truncate (i.e., 13/2 yields 6)
102:    C { II := JJ*0.5; } // truncate (i.e., 13*0.5 yields 6)
103:    C { II := 0x7FF_FFFF; } // hex representation of integer
104:    C { II := JJ&0x00ff; } // bitwise and with a hex number
105:    If (II >= 99) {} // conditional expression
106:    Loop II {} // variable loop count
107:    Loop 0xff { } // loop count defined in hex
108: } //end Pattern

```

## 5.8 Logic expressions—*logic\_expr*

A logic expression is used to represent a combination of signal names and signal group names in the context of a design model. Logic expressions are allowed anywhere that the statement syntax calls for a “*logic\_expr*”. The rules for interpretation of this expression are outside of the definition of this standard (see IEE Std 1364-2001 [B2]).

The logic relationship represented by this expression is independent of any timing or waveform considerations. It is solely indicating that the intended function of the design is to combine the named signals and signal groups according to the specified logic relationship.

The allowed operators in a logic expression are as follows:

( )	parentheses
~	bit-wise negation
&	bit-wise and
	bit-wise inclusive or
^	bit-wise exclusive or
^~, ~^	bit-wise equivalence

The following example contains a logic expression in the ScanEnable statement. The scan enable condition for the scan chain CHAIN1 is active if signals AA and BB are both true, thereby satisfying the logic expression 'AA&BB'.

```
109: STIL 1.0 { Design 2005; }
110: Header {
111:   Source "STD 1450.1-2005";
112:   Ann {* sub-clause 5.8 *}
113: }
114:
115: Signals { AA In; BB In; SI1 In; SO1 Out;}
116:
117: ScanStructures S1 {
118:   ScanChain CHAIN1 {
119:     ScanLength 100;
120:     ScanCells CC[1..100];
121:     ScanIn SI1;
122:     ScanOut SO1;
123:     ScanEnable AA&BB;
124:   }
125: }
```

## 5.9 Real expressions—*real\_expr*

A real expression is an expression that evaluates to a real number as determined by the context (i.e., key word) or the variable type on the left-hand side of an expression. Real expressions are allowed anywhere that the statement syntax calls for a “*time\_expr*” or “*real\_expr*”; i.e., anywhere that an engineering unit value is allowed, then a real expression is allowed. Real variables and expressions are defined in a Spec block in STIL.0. A real number can be expressed in one of two formats as follows:

- A real number can be of the form: <number>e<+|-><number>. A real number can be used to represent values that are not standard SI units, for example, a slew rate in volts/nanoseconds.
- A real number can also be of the form <number><prefix><SI unit>. For example, ‘23ns’ is a time expression (nanoseconds), ‘10uF’ is a capacitance expression (micro-Farads). This generic reference can be used whenever one of the standard unit definitions is allowed.

There are commonly accepted rules with regard to the algebraic combination of values with engineering units. Such rules are not defined in this standard; however, specific tools that implement this standard may require compliance with these rules, and it is good practice to keep units consistent with the usage. For example, if a spec value defines VX to be of type voltage, it can be used as time by expressing it as VX\*(1ns/1V).

The following examples are of real number expressions:

```

126: STIL 1.0 { Design 2005; }
127: Header {
128:   Source "STD 1450.1-2005";
129:   Ann { * sub-clause 5.9 * }
130: }
131:
132: Signals { SIG_NAME InOut; }
133:
134: Spec {
135:   Category CAT {
136:     TIME = 25ns;
137:     REALVAR = 5.5;
138:     VOLTAGE = 2.5V;
139:     WATTAGE = 25mW;
140:     SLEWRATE = 1V/1ns;
141:   } // end Category
142: } // end Spec
143:
144: Pattern PAT {
145:   If (TIME >= 23.5ns+1.5ns/2) { } // time expression
146:   If (WATTAGE >= 5V*2uA) { } // where WATTAGE is of type 'Watts'
147:   If (SLEWRATE' >= 5V/1ns) { } // where SLEWRATE is of type 'Real'
148:   If (REALVAR >= 5) { } // where REALVAR is of type 'Real'
149:   If (REALVAR*1mA >= 5mA) { } // where REALVAR is of type 'Real'
150: } // end Pattern
151:
152: Timing {
153:   WaveformTable WFT {
154:     Waveforms { SIG_NAME {
155:       01 { 25ns D/U; } // constant time
156:       01 { (REALVAR*1ns) D/U; } // use a real to define time
157:       01 { (VOLTAGE*(1ns/1V)) D/U; } // use a voltage to define time
158:     } }
159:   }
160: } // end Timing

```

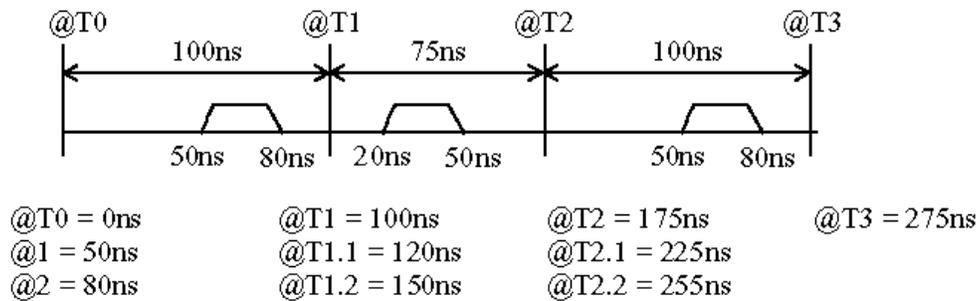
## 5.10 Addition to timing expressions—*time\_expr*

Additional symbols are added for referencing timing events in waveforms. These are additions to the @n symbol (as defined in STIL.0, Subclause 6.13), which allows reference to the timing edges in a timing expression that are in the current period. The new symbols, as defined here (along with the other time referencing capabilities), allow access to period markers and events in subsequent periods. These new facilities are provided to support application in other STIL extensions, such as STIL.3 and STIL.6.

- a) @ => references the time of the prior event (STIL.0).
- b) @n => references the time of the n-th event, where first event is @1 (STIL.0).
- c) @@ => references the time of the current event (STIL.1).
- d) @Tm => references the time of the start of the m-th period, where the current period is numbered 0. The time value returned is relative to T0 of the current period. @T1 represents the end of the current period (i.e., the time relative to the current period for the start of the next period).

- e) @Tm.n => references the events of subsequent periods. The time value returned is relative to T0 of the current period. @T1.3 represents the third event of the second period. Note that events are numbered starting at 1, because the 0-th event is T0 (STIL.1).
- f) EVENT\_LABEL => references a label that is defined by EVENT\_LABEL: anywhere in the current WaveformTable or WaveformDescriptions block (STIL.0).
- g) SPEC\_VARIABLE => references a variable defined in a Spec block (STIL.0).
- h) VARIABLE => references an Integer or IntegerConstant defined in a Variables block (STIL.1).

Figure 1 illustrates events across three periods.



**Figure 1—Referencing timing edges**

### 5.11 SignalVariables—*sigvar\_expr*

A SignalVariable is a variable that is used to hold strings of WFC characters. It can be assigned WFC strings in the same way that signals and signal groups are assigned WFC strings. A *sigvar\_expr* is a reference to a signal variable, with the result being a list of WFCs. A SignalVariable is assigned a list of WFC values, which may then be assigned to actual signals or groups. The list of WFC values contained in a variable of type SignalVariable is obtained by preceding it with \W. A SignalVariable name may be used to pass WFC information as a parameter in a macro or a procedure calls.

SignalVariables can be used in-line by setting them to a list of WFCs in a vector or a condition statement. When used in this way, the data are immediately available for use. Note: See also the definition of formal parameters in the next section. The following example shows in-line use of a signal variable:

```
C { SV = 1111; }
V { GRP = \WSV; }
```

The contents of the SignalVariable are maintained on a call to a procedure and the return; and the content is maintained on the invocation and exit from a macro.

The WFC data in a SignalVariable can be used in much the same way as Signals and SignalGroups. Square brackets are used to indicate the length if there are multiple elements. The following example illustrates typical SignalVariable definitions:

```
Variables {
  SignalVariable SV1; // define a single element variable
  SignalVariable SV3 [0..2]; // define a three element variable
}
```

When used in pattern data, the SignalVariable may be used without the square brackets; in which case, all elements of the SignalVariable are used. If a subset of the elements is desired, then the square bracket notation is used. The length on both sides of the equal sign must be the same, except when the signal-variable is being used as a formal parameter (see 5.12). The following example illustrates typical SignalVariable usage:

```

Pattern
  V { SIG = \W SV1; }
  V { GRP_OF_THREE = \W SV3; }
  V { GRP_OF_TWO = \W SV3[0..1]; }
}

```

The following examples are of SignalVariable usage:

```

161: STIL 1.0 { Design 2005; }
162: Header {
163:   Source "STD 1450.1-2005";
164:   Ann { * sub-clause 5.11 * }
165: }
166:
167: Signals {
168:   BUSX[1..5] In; // defines 5 signals plus a group named BUSX
169: }
170:
171: Variables {
172:   SignalVariable SIG_VAR[1..5];
173: }
174:
175: MacroDefs {
176:   APPLY_VAR {
177:     C { SIG_VAR[1..5] = #; }
178:     V { BUSX = \W SIG_VAR[1..5]; }
179:     // error if above two lines were changed to:
180:     // V { SIG_VAR[1..5] = #; BUSX = \W SIG_VAR[1..5]; }
181:   }
182:   APPLY_TWO_VARS {
183:     C { SIG_VAR[1..5] = #; }
184:     V { BUSX = \W SIG_VAR[1..5]; } // apply 11111
185:     C { SIG_VAR[1..5] = #; }
186:     V { BUSX = \W SIG_VAR[1..5]; } // apply 00000
187:   }
188: }
189:
190: Pattern PAT {
191:   //following macro call use signal variables as parameters
192:   Macro APPLY_VAR { SIG_VAR[1..5] = 11100; }
193:   Macro APPLY_VAR { SIG_VAR[5 4 2 3 1] = 00011; }
194:   Macro APPLY_VAR { SIG_VAR[5..1] = ABBAB; }
195:   Macro APPLY_TWO_VARS { SIG_VAR[5..1] = 11111 00000; }
196:   //following use signal variables in line
197:   C { SIG_VAR[1..5] = 11001; }
198:   V { BUSX = \W SIG_VAR[1..5]; }
199: }

```

## 5.12 Formal parameters in procedures and macros

The concept of a formal parameter was used in STIL.0; however, it was not called that. Now with the introduction of more complicated pattern data expressions, it is necessary to keep the concept of variable usage and formal parameter separate. Whereas a signal variable may be set to a value that becomes immediately available, a parameter in a macro or procedure is only available when it is referenced within the referenced macro or procedure by the use of the # or % characters. It is the same behavior as for any Signal or SignalGroup that is used as a parameter as defined in STIL.0. See the following example:

When the name of a SignalVariable is used as a parameter in a macro or a procedure call:

```
Macro MAC { SV = 1111; }
Call PROC { SV = 1111 0101; } )
```

then the WFC data are not available until a # or % operator is used inside the procedure or macro:

```
MAC {
  C { SV=#; } V { GRP=\WSV; }
}
PROC (
  C { SV=#; } V { GRP=\WSV; }
  C { SV=#; } V { GRP=\WSV; }
)
```

## 5.13 Integer lists—*integer\_list*

An *integer\_list* is used to specify an ordered list of integer values. Integer lists are allowed anywhere that the statement syntax calls for an “*integer\_list*.” The only allowed operator in an integer list is the integer range operator “..”, which specifies a range of integers. An *integer\_list* may contain either single integers, whitespace-separated lists of integers, or integer ranges. IntegerConstants may appear in place of integers. An integer range is represented by two integers (or IntegerConstants) separated by “..”. For example, ‘3..6’ is equivalent to ‘3 4 5 6’ and ‘4..2’ is equivalent to ‘4 3 2’. Whitespace characters are allowed between the integer or IntegerConstant, and the “..” operator. The following code shows examples of integer lists:

```
200: STIL 1.0 { Design 2005; }
201: Header {
202:   Source "STD 1450.1-2005";
203:   Ann {* sub-clause 5.13 *}
204: }
205: Signals {
206:   SIG[1..5] In;
207:   SI1 In { ScanIn 100; }
208: }
209: ScanStructures S1 {
210:   ScanChain CHAIN1 {
211:     ScanLength 100;
212:     ScanCells CC[1..100];
213:   }
214: }
215: Pattern PAT {
216:   V { SIG[5 4 3 2 1] = 11001; }
217:   V { SIG[5..1] = 00110; }
218: }
```

## 6. Statement structure and organization of STIL information

This standard defines additional top-level STIL blocks: Environment. It is delineated in Table 8, which is provided to be complete with Table 8 in STIL.0.

**Table 8—Additions to optional top-level statements**

Statement	Purpose
Environment Clause 18	The Environment block defines relationships of STIL data to external environments. Environment blocks, if referencing other STIL data, shall be defined after the blocks that define that STIL data, unless those references are to information contained in MacroDefs, Procedures, or Pattern blocks, which are allowed to be forward-references. This is always a top-level block. Environment blocks, if present, do not have a defined order with respect to other STIL blocks defined in STIL.0. Environment blocks may appear any place a top-level STIL statement is allowed.
Pragma Clause 19	The Pragma block allows for embedding application-specific code directly into a STIL file/stream. The block may be used as a top-level block or embedded in another STIL block.
PatternFailReport Clause 20	The PatternFailReport block is typically not part of a STIL file/stream that defines a test program; however, as a top-level block, it would be allowed to exist without conflict with other STIL blocks. This block is to be generated by the execution of a STIL test program to report results of execution.

## 7. STIL statement

This clause defines extensions to STIL.0, Clause 8.

The STIL statement identifies the primary version of STIL.0 information contained in a STIL file and the presence of one or more standard Extension constructs. The primary version of STIL is defined in STIL.0.

The extension to the STIL statement allows for a block containing extension identifiers that allow for additional constructs in the STIL file. There may be multiple Extension statements present, to identify the presence of multiple extension environments. The extension name and the extension statements are defined in the individual documents for those standards.

“Include” files are required (per STIL.0) to start with the STIL statement. The extension context as defined in an “Include” file shall be a subset of the definition in the base file; i.e., it is not permissible to specify an extension in an included file that is not already allowed by the base file. One qualifier to this rule is the "IfNeed" option. If the parser does not actually consume the "Include," then the extensions in the file are not relevant.

All other constructs and restrictions for STIL.0, Clause 8 are in effect here.

## 7.1 STIL syntax

```
STIL IEEE_1450_0_IDENTIFIER { (1)  
    ( EXT_NAME EXT_VERSION; )+ (2)  
} // end STIL
```

- (1) **STIL**: A statement at the beginning of each STIL file.  
IEEE\_1450\_0\_IDENTIFIER: The primary version of STIL, as identified by 1.0.
- (2) **EXT\_NAME**: The name of the Extension. This standard is identified by the name **Design**.  
**EXT\_VERSION**: The version of the extension. This standard is identified by the value **2005**.

## 7.2 STIL example

```
219: STIL 1.0 {  
220:     Design 2005;  
221:     DCLevels 2002;  
222: }  
223: Header {  
224:     Source "STD 1450.1-2005";  
225:     Ann {* sub-clause 7.2 *}  
226: }  
227:
```

## 8. UserKeywords statement

This clause defines extensions to STIL.0, Clause 11.

This clause defines additional locations in a STIL file/stream where the UserKeywords statement is allowed to appear.

The UserKeywords construct is expanded from STIL.0 to allow the UserKeywords statement to be defined within any STIL block. When a UserKeywords statement is defined within a STIL block, those definitions shall apply only within that block and contained sub-blocks. It allows Userkeywords to be “locally scoped” to a containing STIL block. Note: STIL.0 allows for UserKeywords only at the top level.

All other constructs and requirements for STIL.0, Clause 11 are in effect here.

### 8.1 UserKeywords syntax

As defined in STIL.0.

### 8.2 UserKeywords example

```
228: STIL 1.0 { Design 2005; }  
229: Header {  
230:     Source "STD 1450.1-2005";  
231:     Ann {* sub-clause 8.2 *}  
232: }  
233:  
234: Signals { A[1..99] InOut; }  
235: SignalGroups { ALLSIGNALS = 'A[1..99]'; }
```

```

236:
237:   /* The UserKeywords construct is the same as defined in STIL.0,
238:      with the additional capability to be defined within another STIL block */
239:
240: Timing ONE {
241:   UserKeywords STARTUP SHUTDOWN;
242:   WaveformTable ONE {
243:     STARTUP { ALLSIGNALS; }
244:   }
245: }

```

## 9. Variables block

This clause defines variables and named constants. If the block is unnamed, then all definitions shall be globally available to all other blocks in the STIL file/stream. If the block has a name, then that name must be referenced by the Pattern, PatternBurst, or Timing block for the variables to be available. Variables and constants that are defined in the global Variables block shall not be overridden in a named Variables block.

This block should be considered in light of the Spec block that also can define constants. The purpose and usage of these two blocks is sufficiently different to warrant their separation. The Spec block is intended to contain parametric data used in specifying device/test characteristics. The Spec block has special handling defined for Category, Min/Typ/Max/Meas, and Selector, which are to facilitate a flexible definition of these test parameters.

The Variables block is used to contain control variables and constants. Typical uses for these constants and variables is to control flow of execution, to provide ‘aliasing’ of values to meaningful names with constants, and to provide run-time parameters.

Although complex expressions may be supported by an ATE system, in general, most ATE systems will be able to support only a limited subset of the design capabilities defined herein. The full capabilities are intended for use by simulation applications or pattern translation tools. See Annex O for more information.

### 9.1 Variables block syntax

```

Variables (VARIABLES_DOMAIN) { (1)
  ( IntegerConstant CONST_NAME := INTEGER ; ) * (2)
  ( Integer VAR_NAME ; ) * (3)
  ( Integer VAR_NAME {
    ( Usage Test ; )
    ( InitialValue integer_expr ; )
  } ) * // end Integer
  ( SignalVariable VAR_NAME ; ) * (4)
  ( SignalVariable VAR_NAME {
    ( Base < Hex | Dec > WFC_LIST ; )
    ( Alignment < MSB | LSB > ; )
    ( InitialValue vec_data ; )
  } ) * // end SignalVariable
  ( WFCConstant CONST_NAME = WFC_LIST ; ) * (5)
} // end Variables

```

- (1) **Variables:** This block contains the definitions of variables and constants. It may be either named or unnamed. If the block is named, then the name must be referenced in the PatternBurst block or Timing Block for the variables to be available.
- (2) **IntegerConstant:** This statement allows the definition of a named integer that has a constant value. The “INTEGER” value on the right-hand side may be either a literal integer value (as defined in

STIL.0, Subclause 6.12) or an integer-expr containing only literal integers and other integer-constant names. The named constant shall be used anywhere that a literal integer value is allowed. An integer-constant can be more widely used than an integer-variable; for example, if K is an integer-constant, then it may be used in defining an indexed list of signals (SIGS[1..K] In;), as a length indicator (ScanIn K; DataBitCount K; ScanOutLength K;), and as a loop counter (Loop K {} MatchLoop K {}).

- (3) **Integer:** This statement or block defines an integer-variable. If the integer-variable is specified as a block, then the following optional attributes may be specified:

**Usage Test:** This statement specifies that the integer-variable is to be used in the translation, load, or execution of the STIL file/stream for an ATE system. Integer-variables without this statement may safely be ignored for the purposes of pattern execution. The default is NO; i.e., do not use as a test variable.

**InitialValue:** This attribute allows the specification of the initial value that is to be assigned to the integer-variable at the onset of highest level domain for the variable, i.e., global = upon creation; PatternBurst = each execution of the burst; Pattern = each execution of the pattern. The value of the expression defined in the InitialValue statement is not established until a context is established in which this integer-variable is in scope. The default initial value is zero.

- (4) **SignalVariable:** This statement or block defines a variable that may be used wherever a signal or group may be used. For example code, see Annex B.

When declaring signal-variables, either the single WFC entity form may be specified that requires no square brackets or the multiple WFC entity form may be specified with the square bracketed notation. The square brackets accomplish two purposes: defining the length of the signal-variable and defining the index range of the signal-variable.

When using a signal-variable within pattern data, it is an error to access a bracketed index that is outside the declared range of a bracketed signal-variable. A signal-variable consumes one WFC for each index when assigned a WFC list. When declared as a bracketed signal-variable, a reference to that signal-variable without brackets is equivalent to a reference of that signal-variable across the declared bracketed range of that signal-variable.

If the SignalVariable is a block, then the following optional attributes may be specified:

**Base <Hex | Dec> WFC\_LIST:** This attribute defines the mapping to WFC characters from hex or decimal values, as defined for SignalGroups in STIL.0, Clause 14. If no base is specified, the default is to use waveform characters.

**Alignment:** This attribute is used to specify whether the mapping from hexadecimal to the signal-variable is to align with the MSB or the LSB. The default is MSB to accommodate the convention used for scan data. This attribute applies only to hexadecimal values because decimal values are always LSB aligned.

**InitialValue:** This attribute allows the specification of the initial value that is assigned to the signal-variable at the onset of each execution of the first or highest-level PatternBurst that uses this signal-variable. Signal-variables declared with no InitialValue are undefined until assigned a value (written to), and it is an error to reference the signal-variable while in the undefined state. The value of the expression defined on the InitialValue statement is not established until a context in which this signal-variable is in scope is initiated. This context occurs under a PatternBurst that references the Variables block that contains this signal-variable, in combination with the PatternExec that invoked this PatternBurst.

- (5) **WFCConstant:** This statement allows the definition of a named list of waveform characters that has a constant value. The named constant shall be used anywhere that a WFC or WFC\_list is allowed.

## 9.2 Variables example

```

246: STIL 1.0 { Design 2005; }
247:   Header {
248:     Source "STD 1450.1-2005";
249:     Ann {* sub-clause 9.2 *}
250:   }
251: Variables {
252:   IntegerConstant BUSTOP := 15;
253:   Integer H1;
254:   Integer H2 { Usage Test; InitialValue 13; }
255:   IntegerConstant RED:=0;
256:   IntegerConstant GREEN:=1;
257:   IntegerConstant BLUE:=2;
258:   Integer COLORS; // Values = RED, GREEN, BLUE
259:   SignalVariable VW; // a single WFC variable
260:   SignalVariable VX[1..5]{ InitialValue 11000; }
261:   SignalVariable VY[7..1]{ Base Hex AB; InitialValue FE; }
262:   WFCConstant RESET=00;
263:   WFCConstant RUN=01;
264:   WFCConstant EXTEST=10;
265:   WFCConstant INTEST=11;
266:   SignalVariable VZ[0..1]; // Values = RESET, RUN, EXTEST, INTEST
267:   SignalVariable FOO[1..6];
268: }
269: Signals {
270:   BUS1[ BUSTOP .. 0 ] Inout;
271:   SIG1 In;
272:   SIG2 In;
273:   SIG[2..6] In;
274: }
275: SignalGroups {
276:   GRP = 'SIG[2..6]';
277: }
278: Procedures {
279:   PROC {
280:     C { FOO=#; VW=#; VX=11111; VY=\h82; VZ=#; } // VY = BAAA AAB
281:     V { SIG1=\W FOO[1]; GRP=\W FOO[2..6]; }
282:     V { SIG2=\WVW; SIG[5..6]=\WVZ; }
283:   }
284: }
285: Pattern P {
286:   Call PROC { FOO=111000; VW=P; VZ=\WRESET; }
287:   Call PROC { FOO=010101; VW=P; VZ=\WRUN; }
288: }

```

## 9.3 Variables scoping

Care should be taken when defining Variables blocks as to whether the variables and constants are to be shared or unique to a pattern. Shared (global) variables are useful for passing data, whereas nonshared (local) variables provide independent operation. The scope of a variable or constant is determined by (1) whether the Variables block is named or unnamed and (2) how a named Variables block is referenced.

The example below shows the various usage models. This example uses integer-variables, but the same scoping rules apply to signal-variables, integer-constants, and WFC-constants.

Variables and constants declared in an unnamed Variables block are global in scope and may be accessed from any pattern context unless that variable is hidden by a declaration of the same name from a named Variables block referenced from a PatternBurst context. Global variables are set to the InitialValue at the start of a PatternExec and maintain the last-assigned value throughout the execution of the PatternExec.

Variables declared in named Variable blocks are local to the Pattern or PatternBurst context where that named Variable block is referenced. Local variables are assigned their InitialValue when the first Pattern that contains that Variables block reference starts to execute, and the value of that variable only persists while the Pattern or PatternBurst context that contains that Variables reference is executing. Each reference of a Variables block in a PatternBurst context defines separate and unique instances of local variables for that context.

STIL.0, Subclause 6.16 defines the rules for name resolution of signals and signal-groups across named and unnamed domains. Because variables exist in the same name space as signals and signal-groups, the same rules apply to the naming of variables. This is illustrated in the following example, where there is a signal named FOO and a signal-variable named FOO that overrides the global signal name.

```
289: STIL 1.0 { Design 2005; }
290:   Header {
291:     Source "STD 1450.1-2005";
292:     Ann { * sub-clause 9.3 * }
293:   }
294:   Signals {
295:     FOO In; BAR Out;
296:   }
297:   SignalGroups GROUPS {
298:     GRP = 'FOO+BAR';
299:   }
300:   Variables {
301:     Integer GLOBAL_VAR;
302:   }
303:   Variables SHARED {
304:     Integer SHARED_VAR {InitialValue 0;}
305:     SignalVariable FOO[1..4];
306:   }
307:   Variables LOCAL {
308:     Integer LOCAL_VAR;
309:   }
310:
311:   Procedures {
312:     // This procedure can only be used in PAT2 or PAT3 because it
313:     // contains a reference to LOCAL_VAR that is defined only for
314:     // those pattern contexts.
315:     PROC1 {
316:       If (GLOBAL_VAR == 100) {} // true
317:       If (SHARED_VAR == 200) {} // true
318:       If (LOCAL_VAR == 2) {} // true when called from PAT2
319:       If (LOCAL_VAR == 3) {} // true when called from PAT3
320:     }
321:   }
322:
```

```

323: PatternBurst {
324:   Variables SHARED;
325:   SignalGroups GROUPS;
326:   PatList { PAT1; }
327:   ParallelPatList {
328:     PAT2 { Variables LOCAL; }
329:     PAT3 { Variables LOCAL; }
330:   }
331: }
332:
333: Pattern PAT1 {
334:   C { GLOBAL_VAR := 100; }
335:   If (GLOBAL_VAR == 100) {} // true
336:   If (SHARED_VAR == 0) {} // true - by initialization
337:   If (LOCAL_VAR == 1) {} // error - not defined in this context
338:   C { SHARED_VAR := 200; }
339:   C { FOO = 1100; } // FOO is a signal variable, not a signal
340:   V { BAR = H; } // BAR is a signal
341:   V { GRP = 1H; } // GRP contains the signal FOO and BAR
342: }
343:
344: Pattern PAT2 {
345:   C { LOCAL_VAR := 2; }
346:   If (GLOBAL_VAR == 100) {} // true
347:   If (SHARED_VAR == 200) {} // true
348:   If (LOCAL_VAR == 2) {} // true
349:   Call PROC1 {}
350: }
351:
352: Pattern PAT3 {
353:   C { LOCAL_VAR := 3; }
354:   If (GLOBAL_VAR == 100) {} // true
355:   If (SHARED_VAR == 200) {} // true
356:   If (LOCAL_VAR == 3) {} // true
357:   Call PROC1 {}
358: }

```

## 9.4 Variables synchronizing

When variables are used, care must be taken as to when the result of a variable expression is available to other expressions. It can apply to multiple expressions in one statement, or it can apply to expressions using variables that are common across parallel patterns.

### 9.4.1 Synchronizing within a Condition or Vector statement

Consider the following case involving two integer variables, FOO and BAR:

```

C { FOO := 1; }
C { FOO := 6; BAR := FOO+1; } // Error in use of FOO

```

After the second C statement, what is the value of BAR? The answer could be either 2 or 7, depending on the implementation. For this reason, it is an error to have the same variable on the left- and right-hand sides of an expression within the same statement. The correct way to express the above is

```
C { FOO := 1; }
C { FOO := 6; }
C { BAR := FOO+1; } //BAR result is 7
```

It is also an error to have a variable on the left-hand side of more than one expression with a statement.

```
C { FOO := 1; FOO := 2; } //Error - multiple assignment to FOO
```

#### 9.4.2 Synchronizing across parallel patterns

Now consider the case of two patterns PAT1 and PAT2 that are executing in lock step and share the common variable FOO:

```
Variables { Integer FOO { InitialValue 0; } } //FOO is globally available
PatternBurst B {
  ParallelPatList LockStep { P1; P2; }
}
Pattern P1 {
  V { SIGS_P1 = 000111; }
  C { FOO := 16; }
  Loop FOO {
    V { CLK1 = P; } // loop 16 times
  }
}
Pattern P2 {
  V { SIGS_P2 = 111000; }
  Loop FOO {
    V { CLK2 = P; } // loop 16 times
  }
}
```

How many times does the loop in pattern P2 execute and why? The answer is 16. The variable FOO is in scope across both patterns because it is a global variable. The C statements are considered to execute in zero time, so the sequence of execution is

- All consecutive C statements are executed in each pattern independently
- Any global variables are exported to other patterns
- The next non-C statement uses the values established from other patterns for all global variables.

## 10. Signals block

This clause defines extensions to STIL.0, Clause 14.

This clause defines additional statements supported within the Signals block. All statements and capabilities as defined in STIL.0, Clause 14 are unchanged.

A new attribute, WFCMap, allows mapping WFC values to other WFC on a signal or group of signals.

When the square bracketed signal notation is used, a signal group by the same name is automatically created.

## 10.1 Signals block syntax

```

Signals { (1)
  ( SIG_NAME < In | Out | InOut | Supply | Pseudo > {
    (WFCMap { (2)
      ( FROM_WFC -> TO_WFC( TO_WFC )+ ; ) * (3)
      ( FROM_WFC1 (FROM_WFC2) -> TO_WFC ; ) * (4)
    } ) // end WFCMap
  } ) *
}

```

(1) **Signals**: Refer to STIL.0 for the definition of the Signals block and statements not defined in this extension.

(2) **WFCMap**: It is an optional attribute block used to define waveform character mapping to be used in the pattern data. See also Clause 11, where identical syntax is used to define WFCMap for signal groups. For example code, see 5.2 and Annex G.

This statement defines the mapping of characters only, not waveforms. The resultant list of WFCs is associated with the waveform tables that are referenced in a pattern.

The WFCMap attribute defines the global mapping of all references to the identified signals (i.e., similar in usage to the Termination attribute as defined in STIL.0). This attribute, if present, shall be defined only once for a signal. The mapping applies to the base signals whether it appears in the Signals or in the SignalGroups block.

Refer to Clause 15 for the definition of usage of the mapped WaveformCharacter values. For example code, see Annex G.

(3) FROM\_WFC -> TO\_WFC(TO\_WFC)+: When there are multiple WFCs on the right-hand side, this defines a mapping to be used for reading back data from the device (see CompareSubstitute operation in 13.2). The waveform referenced by the FROM\_WFC shall contain a Substitute character. The TO\_WFC characters are references to the set of allowed response waveforms. The order of characters in the TO\_WFC list is important; each WaveformCharacter represents the corresponding waveform in the selected Waveforms block. See 15.2 for usage of the mapping constructs in pattern data.

(4) FROM\_WFC1 FROM\_WFC2 -> TO\_WFC : When there is one WFC on the right-hand side, this defines a mapping of the specified WFC into a different WFC (see 15.2 for usage of the mapping constructs in pattern data). This replacement applies when, in a given vector, there is a \m (map) or \j (join) specified in the pattern data. Whitespace is not required when there are two WaveformCharacters (FROM\_WFC1 FROM\_WFC2) and they are not order sensitive.

## 10.2 Signals example

```

359: STIL 1.0 { Design 2005; }
360: Header {
361:   Source "STD 1450.1-2005";
362:   Ann { * sub-clause 10.2 * }
363: }
364: Variables {
365:   IntegerConstant BUS1_TOP := 15;
366: }
367: Signals {
368:   BUS[0..15] In;
369:   // the above statement defines 16 signals named BUS[0], BUS[1], etc.
370:   // the above statement also defines 16 element global signal group named BUS[0..15]
371:   BUS1[ BUS1_TOP .. 0 ] Inout;
372:   DIR In;
373:   A0 In {

```

```
374:      WFCMap {
375:          z->x; // single-WFC mapping
376:          01->x; // two-WFC mapping (requires presence of \j)
377:      }
378:  }
379: }
380: Pattern P {
381:     V { BUS = 00000000 11111111; }
382:     V { BUS[0..15] = 11111111 00000000; }
383:     V { BUS[5] = 0; }
384:     V { BUS[8..15] = 11111111; }
385: }
```

### 10.3 Bracketed signal notation enhancement

This subclause defines enhancements to the naming of signals and groups using square brackets as defined in STIL.0, Subclause 6.10. A further extension is to STIL.0, Clause 14, to automatically create group names for busses. All existing capabilities, as defined in STIL.0, are unchanged.

#### 10.3.1 Use of *integer\_list* in signals and groups

The allowed syntax within the square brackets of a signal or group is extended to include *integer\_list* notation (see 5.13). This capability allows for nonsequential numbering of signals and is particularly useful in mapping one signal, group, or signal variable to another. The following examples illustrate the allowed use of square brackets:

```
BUS [1..15]
ODD [1 3 5 7 9]
BUS [1..10] = \WVAR [1 2 3 4 5 10 9 8 7 6]
BUS [11] = X
BUS [12..15] = 1010
```

#### 10.3.2 Autonomed group for bracketed signal names

When a set of signals is defined in a Signals block using the square bracket notation then, under certain conditions as defined in the following list, a global signal-group by the same name is also defined with the length equal to the range of the index. The following rules apply to this new signal-group definition:

- The indexing order of this new signal-group corresponds to the ordering in the signal definition; i.e., if eight signals are defined as BUS[7..0] In;, the signal-group named BUS is also addressed as BUS[7..0].
- If there is more than one set of signals by the same base name, then no group is defined, i.e., BUS[0..5] and BUS[8..15].
- If a SignalGroups block defines a signal-group by the same base name as a set of signals, then the signal-group definition overrides (according to the standard STIL.0 rules).
- If a set of signals is defined using discrete numbering, then the automatic signal-group is not created; e.g., SIG[1 2 3 4] will not result in a signal-group called SIG, whereas SIG[1..4] will do so.

## 11. SignalGroups block

This clause defines extensions to STIL.0, Clause 15.

This clause defines additional statements supported within the SignalGroups block. All statements and capabilities as defined in STIL.0, Clause 15 are unchanged.

As presented in Clause 10, the WFCMap attribute has been added to SignalGroup declarations in this block. Additional details on the WFCMap can be found in Clause 10 and Clause 15.

Additional rules on the use of indexed signal groups are defined.

### 11.1 SignalGroups syntax

```
SignalGroups { (1)
  ( GROUPNAME = sigref_expr {
    ( WFCMap { (2)
      ( FROM_WFC -> TO_WFC( TO_WFC )+ ; ) *
      ( FROM_WFC1 FROM_WFC2 -> TO_WFC ; ) *
    } ) // end WFCMap
  } ) *
} // end SignalGroups
```

- (1) **SignalGroups:** Refer to STIL.0 for the definition of the SignalGroups block and statements not defined in this extension.
- (2) **WFCMap:** See definition of WFCMap in Clause 10 for details of this attribute.

### 11.2 SignalGroups, WFCMap, and Variables example

```
386: STIL 1.0 { Design 2005; }
387: Header {
388:   Source "STD 1450.1-2005";
389:   Ann { * sub-clause 11.2 * }
390: }
391:
392: Variables {
393:   IntegerConstant HI_INDEX := 5;
394:   Integer COMPLETE_DECL { InitialValue -1; }
395:   SignalVariable H1[0..7] { Base Hex DU; }
396:   WFCConstant FETCH = AABBA { Ann { * command to fetch * } }
397:   WFCConstant STORE = ABBBA { Ann { * command to store * } }
398:   SignalVariable BUS_STATE[1..5] { InitialValue AAAAA; }
399:   WFCConstant GRAB = UUUU UUUU { Ann { * moves cursor * } }
400:   WFCConstant RELEASE = DDDD DDDD { Ann { * releases cursor * } }
401:   SignalVariable CUR_STATE[0..7] {
402:     InitialValue \W RELEASE; // initial value is DDDD DDDD
403:   }
404: }
405:
406: Signals {
407:   TOPBUS[HI_INDEX .. 0] In;
408:   H[0..7] In;
409:   A0 InOut; A1 InOut; A2 InOut; A3 InOut;
410: }
```

```
411:
412: SignalGroups {
413:   GRP1 = 'A+B+C'; // implied indexing is [0..2]
414:   GRP2[0..2] = 'A+B+C';
415:   GRP3[3..1] = 'A+B+C';
416:   BUS_A = 'TOPBUS[HI_INDEX .. 0]';
417:   GRP_A = 'A0+A1+A2+A3' {
418:     WFCMap {
419:       z->x; // single-WFC mapping
420:       01->x; // two-WFC mapping (requires presence of \j)
421:     }
422:   }
423:   BUS_H = 'H[0..7]' { Base Hex DU; }
424: }
425:
426: MacroDefs {
427:   MDATA {
428:     C { H1 := #; BUS_STATE = #; CUR_STATE = #; }
429:     V { BUS_H = \W H1; }
430:     V { BUS_A = \W BUS_STATE; BUS_H = \W CUR_STATE; }
431:   }
432: }
433:
434: Pattern P {
435:   C { BUS_A = AAAAA; GRP_A = 0000; BUS_H = \h00; }
436:   V { BUS_A = \W FETCH; }
437:   V { BUS_H = \W RELEASE; }
438:   Macro MDATA { H1 = 36; BUS_STATE = \W STORE; CUR_STATE = \W GRAB; }
439: }
```

### 11.3 Default WFCMap attribute value

A WFCMap attribute does not have to be present on a signal-group declaration for a WFC-map to be in effect. If a WFCMap on any signal or signal-group declaration contains the same base signals, then that mapping is in effect.

### 11.4 Defining indexed signal groups

A signal-group may be defined with only the base name, or with the square bracket syntax to define the indexing of that signal-group. The following rules apply:

- a) If the indexed form is used, then the size of the signal-group shall match the size of the signal expression on the right-hand side.
- b) If no square bracket index is provided, then [n..0] is assumed; i.e., the right-most signal of the right-hand side expression is indexed as element 0.
- c) It is an error to define two signal-groups with the same base name; i.e., GRP[0..7] and GRP[10..17].

## 12. PatternBurst block

This clause defines extensions to STIL.0, Clause 17

This clause defines additional statements supported within the PatternBurst block. All statements and capabilities as defined in STIL.0, Clause 17 are unchanged.

Two new pattern grouping structures are defined: ParallelPatList and PatSet. Also, Fixed and Extend statements are defined to allow the specification of how multiple patterns are to be executed.

The If and While statements are provided to allow conditional execution of patterns within a burst.

### 12.1 PatternBurst syntax

```

PatternBurst PAT_BURST_NAME { (1)
  ( Variables VARIABLES_DOMAIN; )* (2)
  ( Fixed ) { (cyclized-data)* } ) (3)
  ( PatList { (4)
    ( PAT_NAME_OR_BURST_NAME {
      ( Variables VARIABLES_DOMAIN; )*
      ( If boolean_expr ; ) (5)
      ( While boolean_expr ; ) (6)
    })* // end pat_name_or_burst_name
  })* // end PatList
  ( PatSet { (7)
    ( PAT_NAME_OR_BURST_NAME ; )*
    ( PAT_NAME_OR_BURST_NAME {
      ( Variables VARIABLES_DOMAIN; )*
    })* // end pat_name_or_burst_name
  })* // end PatSet
  ( ParallelPatList ( SyncStart | Independent | LockStep ) { (8)
    ( PAT_NAME_OR_BURST_NAME ; )*
    ( PAT_NAME_OR_BURST_NAME {
      ( Variables VARIABLES_DOMAIN; )*
      ( Extend ; ) (9)
      ( If boolean_expr ; )
      ( While boolean_expr ; )
    })* // end pat_name_or_burst_name
  })* // end ParallelPatList

```

- (1) **PatternBurst**: Refer to STIL.0 for the definition of the PatternBurst block and statements not defined in this extension.
- (2) **Variables**: This statement allows reference to a named block of variables to be allowed by all patterns and pattern bursts within this block (see Clause 9 for the definition of the Variables block).
- (3) **Fixed**: This statement allows the specification of signals for which the operation is to be defined outside of the patterns that make up a pattern-burst. The signals may be set to a fixed static state or may be fixed to a WFC that is to be repeated. This statement shall be associated with the entire pattern burst block and is in effect until another Fixed statement or the end of the PatternBurst block. It requires that the list of signals in the sigref\_expr not be redefined in the associated patterns. If a signal is fixed to a WFC, then that signal shall occur in each pattern (or one of the patterns in the case of a parallel-pat-list), and that WFC shall be defined within each WFT that is selected for each vector. If a static fixed value is desired, then this is accomplished with the \e syntax (see 15.3), which requires no WFC/WFT definitions. For example code, see 12.3.

This statement performs a similar function to the Fixed statement within a pattern. The difference is in the scope. The Fixed statement within a pattern is in effect from its occurrence to the end of the

pattern, whereas the Fixed statement within a PatternBurst is in effect for the entire burst of patterns. Use of Fixed in the static format (e) does not require any change to the pattern.

The assertion within a pattern of a signal that is in a Fixed state may occur, as long as the assertion is to the same state or WFC to which it has been Fixed. Any assertion on a Fixed signal other than that specified in the Fixed statement is an error.

- (4) **PatList:** The PatList block performs exactly the same function as defined in STIL.0. It is repeated here to show the new optional statements that are allowed within a pattern list, namely the If and While attributes.
- (5) **If:** This statement defines a conditional requirement on the execution of the PAT\_OR\_BURST\_NAME; this block will execute only if the *boolean\_expr* evaluates to true. The expression is only evaluated once, before the execution of the referenced pattern or burst.
- (6) **While:** This statement defines an iterative option for execution of the PAT\_OR\_BURST\_NAME; the referenced pattern or burst shall be executed repeated times as long as the *boolean\_expr* evaluates to true. The expression is evaluated each time the referenced pattern or burst starts to execute.
- (7) **PatSet:** This block defines a set of patterns and/or bursts that have no requirement on the order of execution of each reference. This construct is intended primarily for defining data in an interim format before being presented to a tester, to identify a set of patterns that have no external constraints on order of execution. If these data are presented to a tester, then each pattern shall be self-initializing and capable of executing independently.

The PatSet block is similar to the PatList in that it is used to define a list of patterns and/or bursts and the environment for interpreting them. The difference is that the PatSet does not imply any sequencing requirements. Thus, the system integrator is free to take that list of patterns and use them in any sequence desired. All optional statements that are defined in STIL.0 for PatList also are available in a PatSet block.

The requirement on the patterns in a PatSet is that they be self-contained such that any initialization that is required is done within the pattern and not dependent on the execution of a prior pattern. Also, because there is no required ordering of patterns in this block, If and While statements are not supported for these blocks because of ambiguity of expression evaluation.

The PatSet shall not be used to do implicit parallel patterns, even if the set of signals between patterns are nonoverlapping. To do so, it is necessary to define two bursts that contain PatSets and reference them in a higher level with a ParallelPatList.

- (8) **ParallelPatList (SyncStart | Independent | LockStep):** This block defines a set of PAT\_OR\_BURST\_NAME that is to be executed in parallel. Execution of this set of patterns is controlled by optional arguments **SyncStart**, **Independent**, or **LockStep**, as well as the optional statement **Extend**. Parallel patterns do not necessarily run synchronously or finish together. If no arguments are specified to ParallelPatList, then the default operation of the patterns is **Independent**. All optional statements that are defined in STIL.0 for PatList also are available in a ParallelPatList block.

**SyncStart:** This keyword, if present, requires that all PAT\_OR\_BURST\_NAME present in the ParallelPatList block shall start executing at the same moment. During execution, pattern behavior may diverge if patterns contain different Vector counts or different periods in the Vectors.

**Independent:** This keyword, if present, allows each PAT\_OR\_BURST\_NAME present in the ParallelPatList block to start as convenient. This option indicates that the set of patterns executing in parallel have little or no relationship between each other and can be executed independently.

**LockStep:** This keyword is used to specify parallel testing of subdesigns that have independent patterns requiring synchronization throughout the pattern execution. An example of an application in which this is used is

- Situations in which parallel subdesigns have common access constructs that require maintaining the same state on a set of signals for the cores during test (for instance, common wrapper control logic around the subdesigns)

- Parallel testing of subdesigns that have serially connected scan chains
- Mapping patterns onto test equipment that has limited timing flexibility that prevents true independent execution

The details of how the patterns, procedures, and macros are resolved is a function of the tools that create and consume the data. The following general requirements shall be observed for LockStep:

- All patterns shall start at the same time.
  - Each vector in each pattern shall have the same period as the corresponding vector in the other parallel patterns.
  - Each macro and procedure invocation shall occur at the same time and shall be of the same vector length.
  - Signals that occur in multiple patterns shall either be set to the same WFC in every vector or else be controlled by the AllowInterleave statement (see 16.1).
- (9) **Extend:** This statement specifies that the last vector of this pattern may be extended to wait for completion of other parallel patterns. The behavior of Signals during Extend is described in 12.3. Extend is incompatible with LockStep; it is an error to specify Extend in a `pat_or_burst_name` block under a `ParallelPatList` block that defines LockStep. For example code, see 12.3.

If the Extend statement is not used, then the last cycle of a parallel pattern shall complete at exactly the same time as the last cycle of all other patterns that are running in parallel. See information on “tiling” in 12.3.

The behavior of the Signals during the Extend period is determined by the last STIL statement in the pattern. If the last statement in the pattern is a `BreakPoint`, then all Signals will maintain the last asserted state indefinitely until the parallel set of patterns is complete. If the last statement is “`BreakPoint { V { ...} }`”, then the Vectors present in this `BreakPoint` block shall be executed for these Signals until all parallel patterns have completed.

If the last STIL statement in the pattern is not a `BreakPoint` construct, then the last asserted state on all Signals shall be maintained indefinitely as if a `BreakPoint` statement was present in the pattern.

## 12.2 PatternBurst example

The following example illustrates a `PatList` and a `ParallelPatList`. Things to note include use of `IntegerConstant`; use of `If/While` in a burst; use of `InitialValue` and the expectation that a variable initializes upon each execution; and the use of `!` in a boolean expression (see also Annex K).

```

440: STIL 1.0 { Design 2005; }
441: Header {
442:   Source "STD 1450.1-2005";
443:   Ann { * sub-clause 12.2 * }
444: }
445: Variables {
446:   IntegerConstant TRUE := 1;
447:   IntegerConstant FALSE := 0;
448:   Integer PAT1_COMPLETED { InitialValue FALSE; }
449:   Integer PAT3_COMPLETED { InitialValue FALSE; }
450: }
451:
452: PatternBurst BURST {
453:   PatList { // only run PAT2 if PAT1 runs to completion
454:     PAT1;
455:     PAT2 { If (PAT1_COMPLETED); }
456:   }
457:   ParallelPatList { // repeat PAT4 until PAT3 runs to completion

```

```
458:     PAT3;
459:     PAT4 { While (!PAT3_COMPLETED); }
460:   }
461: } //end BURST
462:
463: Pattern PAT1 {
464:   V{ } V{ } V{ }
465:   C { PAT1_COMPLETED := TRUE; }
466: }
467: Pattern PAT3 {
468:   V{ } V{ } V{ }
469:   C { PAT3_COMPLETED := TRUE; }
470: }
```

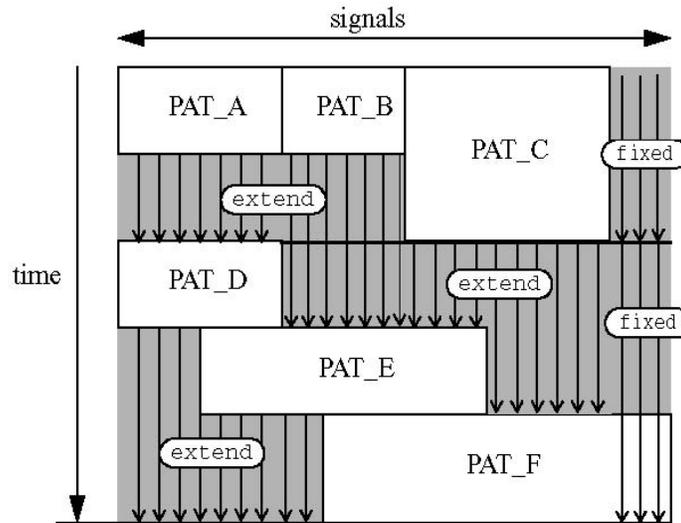
### 12.3 Tiling and synchronization of patterns

Pattern tiling is the process of connecting patterns together end-to-end in time and side-to-side by signal. The following list is a summary of the facilities provided for the purpose of tiling and the rules associated with tiling:

- a) PatList and PatSet allow specification of patterns that run sequentially (i.e., one after the other). Each pattern shall run to completion before the next pattern can begin.
- b) ParallelPatList allows specification of patterns that run at the same time. The starting of the patterns is determined by the keywords, SyncStart, Independent, or LockStep. All patterns shall run to completion before the next block, PatList, ParallelPatList, or PatSet, can begin.
- c) If a pattern ends with a Breakpoint statement, then that pattern can be extended in time as needed when run in a ParallelPatList. There are two forms of Breakpoint defined in STIL.0: “Breakpoint;” (semicolon terminated) and “Breakpoint {...}” (followed by a block containing pattern data). In the case of “BreakPoint;”, the pattern is extended by holding all signals in the final state as long as required. In the case of “Breakpoint {...}”, then that block is repeated as necessary to extend the pattern.
- d) The “Extend;” statement can be placed on any pattern in a ParallelPatList, which effectively specifies that the pattern can be extended. This option allows for extending a pattern without having to modify the pattern file itself. If the pattern ends with a BreakPoint, then the extend operation is done as defined by that statement/block. If the pattern does not end with a BreakPoint statement/block, then the last vector is extended as though the pattern ended with an Extend-semicolon statement.
- e) The extend operation is in effect until a new pattern includes the extended signals or until the end of the outermost PatternBurst.
- f) With regard to synchronizing patterns after a ParallelPatList execution, it is necessary that any pattern that terminates early in a ParallelPatList has one of the above extend mechanisms specified; else it is an error. Also, note that all patterns must terminate or extend to the point in time for the next pattern to execute. If the cyclized extend is being used, care must be taken that the period multiples of each pattern in parallel end up with the same composite time (usually accomplished best when periods of parallel patterns are the same).
- g) Two facilities for defining signals are fixed (i.e., that do not change for the duration of a pattern or a burst of patterns). The Fixed statement may be placed at the top of the PatternBurst; in which case, it is fixed for all patterns within the burst. The Fixed statement can be placed within a Pattern; in which case, it is fixed only for the duration of the pattern. If a set of signals is to be fixed and are not part of any pattern, then a pattern file containing only the Fixed statement is required.

- h) If patterns running in parallel contain common signals, then the activity on these signal shall be coordinated in all patterns that use them. The simple method for coordinating signals (see next point for the other option) is to run the patterns in LockStep and ensure that the same WFT/WFC occurs in every cycle.
- i) If patterns running in parallel contain common signals where the activity differs, then the AllowInterleave statement shall be used. This mechanism requires that only one parallel pattern defines that activity in any given cycle, and every cycle must have one pattern that defines the activity (i.e., there is no automatic repeat into the next vector).

Figure 2 is an example of pattern tiling using the statements available in the PatternBurst. Refer to Clause 12.



**Figure 2—A collection of patterns to be executed**

One example of STIL code that would specify the above action is as follows:

```

471: STIL 1.0 { Design 2005; }
472: Header {
473:   Source "STD 1450.1-2005";
474:   Ann { * sub-clause 12.3 * }
475: }
476: Signals { SIG1 In; SIG2 In; SIG3 Out; }
477:
478: PatternBurst BURST {
479:   Fixed {'SIG1+SIG2+SIG3' =\eUUh;}
480:   ParallelPatList SyncStart {
481:     PAT_A {Extend;}
482:     PAT_B {Extend;}
483:     PAT_C {Extend;}
484:   }
485:   Fixed {'SIG1+SIG2+SIG3' =\eDDX;}
486:   PatList {
487:     PAT_D {Extend;}
488:     PAT_E {Extend;}
489:     PAT_F;
490:   }
491: }

```

This code causes the following actions:

- PAT\_A, pat\_B, and PAT\_C are initiated at the same time. The prior Fixed statement defines a set of fixed signals that are not part of any of the three patterns but must be maintained in the specified state. In this example, SIG1 and SIG2 are held to a static input high state, whereas SIG3 is required to output a constant output high state.
- PAT\_A and PAT\_B are allowed to extend, because they are expected to complete in less time than PAT\_C. Extending means that the state of all signals at the end of the last vector of the pattern can be held, which effectively extends the period of the last vector.
- PAT\_C must run to completion before any further activity in this burst can be started.
- PAT\_D, PAT\_E, and PAT\_F are run sequentially, starting immediately after completion of the last vector of PAT\_C.
- A new Fixed statement defines activity on SIG1, SIG2, and SIG3. All other unused signals in PAT\_D go to their default state.
- After completion of PAT\_D and PAT\_E, signals not defined in the pattern to follow are specified to be extended.
- When PAT\_F executes, the three Fixed signals are required to remain in the defined fixed states.

## 12.4 If and While statements

Conditional PatternBurst statements If and While allow selective execution of a pattern or pattern burst based on the evaluation of a boolean expression.

The following example defines two pattern variables whose scopes are the pattern burst named WITH\_VARS. The variable COUNT has an initial value of 25. The variable RESULT initially has an undefined value and holds the value returned from the user function TESTRESULT().

```
492: STIL 1.0 { Design 2005; }
493: Header {
494:   Source "STD 1450.1-2005";
495:   Ann { * sub-clause 12.4 * }
496: }
497: Variables {
498:   Integer COUNT {InitialValue 25;}
499:   Integer RESULT;
500: } //end Variables
501:
502: Signals {
503:   CLK In;
504:   X[1..16] InOut;
505: }
506:
507: SignalGroups {
508:   SOME_PINS = 'X[1..16]';
509: } //end SignalGroups
510:
511: PatternBurst WITH_VARS {
512:   PatList {
513:     FIRST_PAT;
514:     SECOND_PAT { If (RESULT == 1); }
515:   } //end PatList
516: } //end PatternBurst
517:
```

```

518: // The If statement provides control over the execution of second_pat.
519: // Execution of second_pat depends on the value of the variable RESULT.
520:
521: Pattern FIRST_PAT {
522:   UserFunctions TESTRESULT; //returns a 1 if last executed vector passed
523:   Loop COUNT { V { CLK = P; } }
524:   V { SOME_PINS = 11110000 HHHLLLLL; }
525:   C { RESULT := TESTRESULT(); }
526: } //end Pattern

```

### 13. Timing block and WaveformTable block

This clause defines extensions to STIL.0, Clause 18.

This clause defines additional constructs supported within the waveform statement of a WaveformTable block. All statements and capabilities as defined in STIL.0, Clause 18 are unchanged.

#### 13.1 Additional domain specification

An additional domain specification is added to the Timing block to accommodate the new Variables definition.

**Timing** TIMING\_NAME { (1)  
( Variables VARIABLES\_DOMAIN; )\* (2)

- (1) **Timing:** Refer to STIL.0 for the definition of the Timing block and statements not defined in this extension.
- (2) **Variables:** This statement allows reference to a named block of variables (Clause 9).

#### 13.2 CompareSubstitute operation—s, S

Additional event characters are defined: CompareSubstitute and CompareSubstituteWindow. These events resolve event data with actual response during test generation or read response data from a device. For example code, see Annex I.

Table 9 defines the additional event characters and is incremental to Table 10 of STIL.0.

**Table 9—Compare events**

	Identifier	Icon	Definition
S	CompareSubstitute		Perform a Compare operation at this time, and return the value of the resulting operation as a CompareHigh, CompareLow, CompareUnknown, or CompareOff value.
s	CompareSubstituteWindow		Perform a Compare operation over a period of time, and return the value of the resulting operation as a CompareHigh, CompareLow, CompareUnknown, or CompareOff value. Terminated by a CompareOff event.

The CompareSubstitute event allows stimulus to be defined for device test and supports a mechanism to acquire the device response to this stimulus. How this response information is returned is outside the scope of this standard, but an example is provided (Annex I).

The CompareSubstitute event is expected to be found only once in a waveform definition. If multiple CompareSubstitute characters are present in one waveform definition, then all Compare operations shall return the same value or the CompareUnknown state shall be returned.

The CompareSubstitute event is expected to be defined in conjunction with a WFCMap definition for each WaveformCharacter that represents a waveform containing the CompareSubstitute event. The WFCMap defines what WaveformCharacter values are returned for each supported state value returned by CompareSubstitute. If no WFCMap is defined, or a WaveformCharacter for a specific Compare value is not defined in the WFCMap for this waveform, then the return information is not mapped to waveform references, but explicit events are returned with the vector data, as defined in 15.3.

## 14. ScanStructures block

This clause defines extensions to STIL.0, Clause 20.

The STIL.0 syntax is extended to include additional information required for efficient simulation (i.e., eliminating the need to serially simulate load/unload cycles) of scan patterns when the design includes complex scan cells. A “complex scan cell” is defined to be a state element that is loaded/unloaded by a single element of the scan chain data, but it may contain multiple states internally. A related operation is that of loading multiple state elements from multiple elements of a scan chain. For example code, see Annex L.

The STIL.0 syntax is also extended to support scan segments and scan groups. For more information, see 14.6.

All other constructs and requirements for STIL0, Clause 20 are unchanged.

### 14.1 ScanStructures syntax

```

ScanStructures (SCAN_STRUCT_NAME) { (1)
  ( InheritScanStructures SCAN_STRUCT_NAME ; ) (2)
  ( ScanChain CHAIN_NAME {
    ScanLength integer_expr ; (3)
    ( ScanOutLength integer_expr ; ) (4)
    ( ScanEnable logic_expr ; ) (5)
    ( ScanCellType (CELL_TYPE_NAME) { (6)
      ( ( If boolean_expr ) CellIn INTERNAL-REF-LIST ; ) * (7)
      ( ( If boolean_expr ) CellOut INTERNAL-REF ; ) *
    } ) *
    ( ScanCells { (8)
      ( cell_ref (CELL_TYPE_NAME) ; ) *
    } ) // end ScanCells
    ( Base Hex ; ) // optional for cell groups (9)
    ( Alignment <MSB|LSB>; ) // optional for cell groups (10)
  } ) * // end ScanChain
  ( ScanChainGroups { (11)
    ( GROUP_NAME { (GROUP_OR_CHAIN_NAME; ) * } ) *
  } ) // end ScanChainGroups
} // end ScanStructures

```

- (1) **ScanStructures**: Refer to STIL.0 for the definition of the ScanStructures block and statements not defined in this extension.

- (2) **InheritScanStructures** *SCAN\_STRUCT\_NAME*: This statement allows reference to another scan structure block that is to be included into the current block. All scan chains, scan cells, and scan cell types in the referenced block become part of the current block. Local definitions shall override definitions in an inherited scan structure.
- (3) **ScanLength** *integer\_expr*: As defined in STIL.0, with the extension (a) support for integer expressions and (b) made optional; in which case, the length is determined by the length of the shift data.
- (4) **ScanOutLength** *integer\_expr*: As defined in STIL.0, with the extension to support integer expressions.
- (5) **ScanEnable** *logic\_expr*: This optional statement allows designation of a single Signal or a complete expression to represent the design constraints, if any, necessary to allow access to the scan shift operations for this scan chain. See example usage in Annex C.
- (6) **ScanCellType** (*CELL\_TYPE\_NAME*): In the new brace delimited form of specifying scan cells, an optional cell type name may be defined. The cell type may be unnamed; in which case, it applies to all scan cells without an explicit name reference. The cell type can be named; in which case, it applies to all scan cells with the corresponding name. Cell types are valid only in the context of the current scan structure block or any block that inherits this block. The operation of the cell is defined by the CellIn and CellOut statements that are contained within the block. If no statements exist within the ScanCellType statement, then no scan data are to be consumed by that cell (i.e., the case of a lockup latch).
- (7) (**If** *boolean\_expr*): Optional conditional clause on the CellIn and CellOut statements. The value of *boolean\_expr* is evaluated as necessary by the application to determine the appropriate activity of the scan cell. When *True*, the following CellIn or CellOut statements are applied. See 5.6 for information about *boolean\_expr*:

INTERNAL-REF is a name, and INTERNAL-REF-LIST is a list of names of internal netlist elements separated by whitespace. Internal netlist elements can be internal design nets or scan cell names (*cell\_ref*). Inversion is indicated by inserting the "!" character before or after names. When CellIn and CellOut constructs are inherited through InheritScanStructure constructs, the names of all inherited netlist elements are prefixed with the INST\_NAME (and a period) to identify specific instances of these state element names. All references to netlist elements (INTERNAL-REF, INTERNAL-REF-LIST, and *cell\_ref*) must be within the name space recognized by the STIL interpreter (e.g., simulator) to enable parallel simulation.

**CellIn**: This optional statement indicates that the nets in INTERNAL-REF-LIST are to be loaded with the data value corresponding to the current scan cell, with possible inversion as indicated by a "!" character. A "!" indicates inversion between the input of the scan cell and the optional name after the "!". If no name follows the "!", then the inversion is inside the named scan cell, between the cell input and the state element.

**CellOut**: This optional statement indicates that when the *boolean\_expr* is true, the INTERNAL-REF is to be unloaded (into the scan cell name) with the data value corresponding to the current scan cell, with possible inversion as indicated by a "!" character. A "!" indicates inversion between the name preceding the "!" and the output of the scan cell. If no name precedes the "!", then the inversion is inside the named scan cell, between the state element and the cell output.

Both **CellIn** and **CellOut** statements may be qualified by **If** *boolean\_expr*. It indicates that the nets in the **CellIn** and **CellOut** statements are only to be considered in a pattern where the *boolean\_expr* is *True*. See Table 10 for valid use cases for **CellIn** and Table 11 for valid use cases for **CellOut**:

**Table 10—Use cases for CellIn**

No CellIn statement	No information available about scan cell load operations. Full simulation is required.
CellIn statement	The parallel load operation is defined and consistent. No load operations are to be simulated.
<i>boolean_expr</i> CellIn statements	The parallel load operation is defined for the case when the boolean expr is true. When all <i>boolean_expr</i> are false, the load operation must be evaluated by serial simulation. It is an error if more than one boolean expr evaluates to true.
Both simple CellIn and <i>boolean_expr</i> CellIn statements	The load operation is fully defined. If all boolean expr are false, then the simple cell in block is to be executed.

**Table 11—Use cases for CellOut**

No CellOut statement	No information available about scan cell unload operations. Full simulation is required.
CellOut statement	The parallel unload operation is defined and consistent. No load operations are to be simulated.
<i>boolean_expr</i> CellOut statements	The parallel unload operation is defined for the case when the boolean expr is true. When all <i>boolean_expr</i> are false, the unload operation must be evaluated by serial simulation. It is an error if more than one boolean expr evaluates to true.
Both simple CellOut and <i>boolean_expr</i> CellOut statements	The unload operation is fully defined. If all boolean expr are false, then the simple cell in block is to be executed.

- (8) **ScanCells:** This statement shall appear at most once in a scan chain block. It is used to define the ordered list of scan-cell-names associated with a scan-chain. See 14.2 for the for the allowed formats for a *cell\_ref*. The block form of this statement allows for reference to a scan type block that defines complex scan cells. For example code, see Annex L.
- (9) **Base Hex:** This statement is optional and is used when the referenced list of scan cells is to be used as a cell group. The only attribute allowed is Hex. Note that this does not control the format of data for scan shifting, which is specified in the ScanIn/ScanOut signal definition. See also the statement Alignment. For information about cell groups, see 14.6.
- (10) **Alignment <MSB|LSB>:** This statement is optional and is used when the referenced list of scan cells is to be used as a cell group. This statement is only applicable when data are specified in hex and specifies whether the most significant bit of the hex data is to be aligned with the first cell of the chain (MSB) or whether the least significant bit of the hex data is to be aligned with the last cell of the chain (LSB). For information about cell groups, see 14.6.
- (11) **ScanChainGroups:** A scan chain group is a shorthand way for specifying a set of scan chains. The names that comprise a group shall be chain names or other group names, and they shall be defined either in the global ScanStructures block or the current named ScanStructures block in effect. The named groups can be referenced by the ActiveScanChains statement in a macro or procedure to specify the active scan chains for a shift operation (see 14.3 and 14.5).

## 14.2 Scan cell naming—*cell\_ref*, *chain\_ref*, *cell\_group*, *chain\_group*

The naming and grouping of scan cells involves the concepts as defined here. Refer to 14.1 for the definition of scan cell definition statements.

- a) *cell\_ref*: A scan cell reference is one of the following:
  - 1) A scan cell name.
  - 2) A set of scan cell names of the form name[...].
  - 3) A scan chain name, as determined by the global ScanStructures block or the current named ScanStructures block in effect. Note that a scan chain name may refer to either a complete scan chain, a chain segment, or a cell group (per 14.6).
- b) *chain\_ref*: The name of an individual scan chain that is constructed of an ordered sequence of scan cells. A scan chain is defined in a ScanStructures block. Unlike scan cell names (and like signal group names), scan chain names are only valid when they are in scope. A scan chain name is in scope if
  - 1) It is defined in the global ScanStructures block.
  - 2) It is in the currently named ScanStructures block in effect (as determined in a PatternBurst).
  - 3) It is explicitly specified with the domain::name construct.

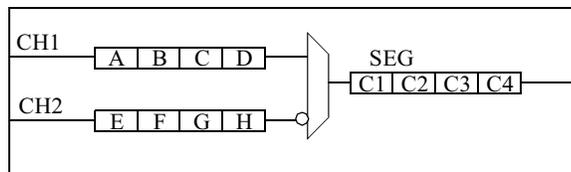
A scan chain comprises names and symbols as follows:

- 1) Scan cell name: following the naming rules of STIL.0, Subclause 6.8.
  - 2) Indexed sequence of names: defined by enclosing the indices in square brackets.
  - 3) Inversion symbol “!”: exclamation point (used without double quotes).
  - 4) Scan chain name: A scan chain name that is defined either in the global scan structures blocks or the current named ScanStructures block in effect.
- c) *cell\_group*: An arbitrary, ordered list of scan cells. A *cell\_group* is defined with the same syntax as a *chain\_ref* and follows the same rules for scoping. The difference between these two concepts is that the *cell\_group* has no requirement for physical connection of the cells. It is intended for either the identification of like cells or the accessing of cells in a simulation operation using direct access to the cell.
  - d) *chain\_group*: A collection of scan chains. A chain group is defined in the ScanChainGroups block inside of a ScanStructures block (Figure 3).

```

ScanStructures {
  ScanChain SEG {
    ScanCells { C1; C2; C3; C4; }
  }
  ScanChain CH1 {
    ScanCells { A; B; C; D; SEG; }
  }
  ScanChain CH2 {
    ScanCells { E; F; G; H; !; SEG; }
  }
}

```



**Figure 3—Referencing shared scan segments**

The following code shows an example of each form of syntax:

```
527: STIL 1.0 { Design 2005; }
528: Header {
529:   Source "STD 1450.1-2005";
530:   Ann { * sub-clause 14.2 * }
531: }
532: ScanStructures STRUCT1 {
533:   ScanChain CHAIN1 {
534:     ScanCells { CXX[1..99]; }
535:   }
536: }
537: ScanStructures STRUCT2 {
538:   InheritScanStructures STRUCT1;
539:   ScanChain CHAIN2 {
540:     ScanCells { CYY[1..99]; }
541:   }
542:   ScanChain CHAIN3 {
543:     ScanCellType MS { CellIn MASTER ! SLAVE; }
544:     ScanCells {
545:       CELL1;
546:       CELL2 MS;
547:       CELL3; !;
548:       CELL4;
549:       CHAIN1;
550:       CHAIN2;
551:     }
552:   }
553: }
```

### 14.3 Scoping rules for ScanStructure blocks

A ScanStructues block may be either un-named (i.e., anonymous or global) or it may have a domain name. These blocks follow similar domain rules to the SignalGroups, Procedures, and MacroDefs blocks. The scoping rules are as follows:

- a) There may be a global (anonymous) ScanStructures block. The global block is optional, and there shall be only one.
- b) Any number of ScanStructures blocks with domain names are allowed.
- c) The ScanStructures domain to be used by a procedure or macro is specified in the PatternBurst block, which thereby allows the reuse of patterns. The statement to select a scan structures domain is "ScanStructures SCAN\_STRUCT\_NAME;".
- d) Scan chain names and scan chain group names exist in the same name space, and hence there are no restrictions between addressing of chain names and group names.
- e) Chain or group names in a global ScanStructures block may be overridden by the definition in a named ScanStructures domain; in which case, the definition in the named domain is used and the global definition is ignored.
- f) Multiple named domains may be selected; in which case, all chain and group names must be unique across all selected domains.

## 14.4 Example indexed list of scan cells

The ScanCell naming constructs allow cells to be expressed as a list using the same notation defined for Signals and SignalGroups in STIL.0, Subclause 6.10. An example is shown as follows. Note that the presence of inversion between cells requires multiple sets of cellnames to be present to represent the inversion at the correct point in the list:

```

554: STIL 1.0 { Design 2005; }
555: Header {
556:   Source "STD 1450.1-2005";
557:   Ann { * sub-clause 14.4 * }
558: }
559:
560: Signals { SI1 In; SO1 Out; }
561:
562: ScanStructures {
563:   ScanChain CHAIN1 {
564:     ScanLength 100;
565:     ScanIn SI1;
566:     ScanOut SO1;
567:     ScanCells AA[1..50] ! AA[51..100];
568:   } //end ScanChain
569: } //end ScanStructures
570:
571:   /* Once a scan chain is defined as an indexed list, then it can
   be referenced in statements that need to access these cells, as shown in
   the following example. Note: This example is taken from IEEE P1450.6, but
   because it is within an Environment block, it should be ignored by a IEEE
   1450.1 parser and processed only by a IEEE 1450.6 parser. */
572: Environment {
573:   CTL {
574:     Internal {
575:       SIG[5..9] {IsConnected In {StateElement Scan CHAIN1 AA[23..27];} }
576:       SIG[12..22] {IsConnected In {CoreSignal COREX AA[55..65];} }
577:     } //end Internal
578:   } //end CTL
579: } //end Environment

```

## 14.5 Example of ScanChainGroups and ActiveScanChain

The following example uses the ScanChainGroups statement in the ScanStructures block to indicate chains that operate together, and it uses the ActiveScanChains statement in the Procedures block to select which of the groups is being used:

```

580: STIL 1.0 { Design 2005; }
581: Header {
582:   Source "STD 1450.1-2005";
583:   Ann { * sub-clause 14.5 * }
584: }
585:
586: Signals {
587:   CLK1 In; CLK2 In; SE In; MODE In; MCLK In;
588:   SI1 In { ScanIn; } SI2 In { ScanIn; }
589:   SO1 Out { ScanOut; } SO2 Out { ScanOut; } SO3 Out { ScanOut; }

```

```
590: }
591: SignalGroups {
592:   _SI1='SI1'; _SO1='SO3';
593:   _SI2='SI1+SI2'; _SO2='SO1+SO2';
594:   _SI3='SI1+SI2'; _SO3='SO1+SO2';
595: }
596:
597: ScanStructures ALL {
598:   ScanChain A_1 { ScanIn SI1; ScanOut SO3; }
599:   ScanChain B_1 { ScanIn SI1; ScanOut SO1; }
600:   ScanChain B_2 { ScanIn SI2; ScanOut SO2; }
601:   ScanChain C_1 { ScanIn SI1; ScanOut SO1; }
602:   ScanChain C_2 { ScanIn SI2; ScanOut SO2; }
603:   ScanChainGroups {
604:     GROUP_A { A_1; }
605:     GROUP_B { B_1; B_2; }
606:     GROUP_C { C_1; C_2; }
607:   }
608: }
609:
610: Procedures {
611:   LOAD_UNLOAD {
612:     C {
613:       CK1=0; CK2=0; MCLK=0; SE=1; MODE=0;
614:       SI1=0; SI2=0; SO1=X; SO2=X; SO3=X;
615:     }
616:
617:     ActiveScanChains GROUP_B;
618:     V { MODE=0; MCLK=P; } // chain selection
619:     V { MODE=1; MCLK=P; } // chain selection
620:     V { MCLK=0; }
621:     Shift {
622:       V { _SI2=##; _SO2=##; CK1=P; }
623:     }
624:
625:     ActiveScanChains GROUP_A;
626:     V { MODE=0; MCLK=P; } // chain selection
627:     V { MODE=0; MCLK=P; } // chain selection
628:     V { MCLK=0; }
629:     Shift {
630:       V { _SI1=#; _SO1=#; CK1=P; }
631:     }
632:
633:     ActiveScanChains GROUP_C;
634:     V { MODE=1; MCLK=P; }
635:     V { MODE=0; MCLK=P; }
636:     V { MCLK=0; }
637:     Shift {
638:       V { _SI3=##; _SO3=##; CK1=P; CK2=P; }
639:     }
640:   }
641: }
```

## 14.6 Scan chain segments and cell groups

The primary application of scan chains is to scan data into or out of a DUT. However, there are two associated uses of this construct also: scan chain segments and cell groups.

A scan chain segment is a ScanChain block that (typically) does not contain a ScanIn or ScanOut statement. The name of a scan chain segment is in the same name space as the scan cells. When the name of the segment is referenced in some other scan chain, then the whole chain (i.e., list of cells) is included in this new chain (see 14.2 for naming rules for cells and cell domains).

A cell group is syntactically identical to a scan chain. The difference being that there is no expectation that the cells are actually connected physically in the design. The grouping is typically used to group cells of like functionality (i.e., an address or data bus). The ordering of cells in a group is typically to indicate logic or arithmetic significance. Two new statements (Base and Alignment) are available to specify the attributes of the data in the cell group. The name of a cell group is in the same name space as the scan cells. The application is responsible for knowing which names are appropriate to use as cell groups.

Complete scan chains are identified by the presence of either a ScanIn or a ScanOut statement in the ScanChain block. All other scan chains are either partial chain segments or cell groups that cannot be accessed directly by a tester.

The following example shows the use of the ScanStructure block to create cell groups:

```

642: STIL 1.0 { Design 2005; }
643: Header {
644:   Source "STD 1450.1-2005";
645:   Ann { * sub-clause 14.6 * }
646: }
647: Signals {
648:   SI In { ScanIn; }
649:   SO Out { ScanOut; }
650: }
651: ScanStructures STRUCT1 {
652:   ScanChain CHAIN1 {
653:     ScanIn SI;
654:     ScanCells { A; B; C[0..63]; D[0..19]; }
655:   }
656: }
657: ScanStructures STRUCT2 {
658:   ScanChain CHAIN2 {
659:     ScanOut SO;
660:     ScanCells { E[23..0]; F; G; H; }
661:   }
662: }
663: ScanStructures GROUPS {
664:   ScanChain GROUP1 {
665:     ScanCells { A; B; F; G; H; }
666:   }
667:   ScanChain GROUP2 {
668:     ScanCells { C[0]; D[0]; E[0]; }
669:   }
670:   ScanChain GROUP3 {
671:     ScanCells { A; C[10..15]; }
672:   }

```

```

673:    ScanChain GROUP4 {
674:        ScanCells { A; B; CHAIN2; }
675:    }
676: }

```

## 15. Pattern data

This clause defines extensions to STIL.0, Clause 21.

This clause defines additional capabilities for defining pattern data. All statements as defined in STIL.0, Clause 21 are unchanged.

Table 12 defines the complete set of backslash operators as defined in both STIL.0 and STIL.1. The usage column of this table is the bnf definition of the allowed syntax, where the terms used in the bnf are defined as follows:

*name	As defined in STIL.0, Subclause 6.10 (Note: allows “name” and name[n..m] formats)
integer	As defined in STIL.0, Subclause 6.12, or an integer expression comprising literals and constants
space	Any whitespace character
hex_number	As defined in STIL.0, Subclause 6.12
wfc_list	As defined in STIL.0, Subclause 6.15
wfc_container	< signal_name   signal_group_name   signal_variable_name   wfc_constant_name >
wfc_ref	< wfc_list   \m wfc_list   “\W”wfc_container > term
drive_event	< “D”   “U”   “Z”   “P” >
compare_event	< “L”   “H”   “X”   “x”   “T”   “V”   “I”   “h”   “t”   “v”   “S”   “s” >
expect_event	< “R”   “G”   “Q”   “M” >
unresolved event	< “N”   “S”   “A”   “B”   “F”   “?” >
event_list	(< drive_event   compare_event   expect_event   unresolved event >)+
term	< “ “   “,” >

Per the definitions in Table 12, the following examples are possible combinations:

- Combinations allowed: \mH \m\mH \j\mH
- Combinations NOT allowed: \m\jH

**Table 12—Backslash pattern data operators**

Operator	Result	Std	Function	Usage (defined in bnf)
\d	WFC	STIL.0	decimal representation of WFCs	\d (wfc_list) integer space term
\e	event	STIL.0	list of events	\e event_list term
\h	WFC	STIL.0	hex representation of WFCs	\h (wfc_list) space hex_number term
\j	WFC	STIL.1	join WFCs on two signals	\j wfc_ref term
\l	WFC	STIL.0	length specifier	\l integer space < wfc_ref   \h (wfc_list) space hex_number   \d (wfc_list) integer space   \e event_list > term
\m	WFC	STIL.1	map WFC to WFC	\m wfc_ref term
\r	WFC	STIL.0	repeat list of WFCs	\r integer space < wfc_ref   \h space hex_number   \d integer space   \e event_list > term
\w	WFC	STIL.0	list of WFCs	\w wfc_list term
\C	event	STIL.1	return list of compare events	\C wfc_container term
\D	event	STIL.1	return list of drive events	\D wfc_container term
\E	event	STIL.1	return list of expect events	\E wfc_container term
\S	WFC	STIL.1	return substitute WFCs (i.e., read back)	\S wfc_container term
\U	event	STIL.1	return list of compare events	\U wfc_container term
\W	WFC	STIL.1	return list of WFCs	\W wfc_container term

### 15.1 Data content read back—\C, \D, \E, \S, \U, \W

This subclause defines syntax for manipulating pattern data by providing read back of the contents of signals, signal groups, variables, and constants. The readback function is of the form:

```
\readback_function SIGNAL_NAME
\readback_function SIGNAL_GROUP_NAME
\readback_function SIGNAL_VARIABLE_NAME
\readback_function WFCCONSTANT_NAME
```

The allowed values for *readback\_function* are

\C SIGNAL-OR-GROUP-NAME—return the last compare event. In the case of a group, return a string of the last compare event for each signal of the group. If no compare event has been established, then return '~' (the tilde character).

**\D SIGNAL-OR-GROUP-NAME**—return the last drive event. In the case of a group, return a string of the last drive event for each signal of the group. If no drive event has been established, then return ‘~’ (the tilde character).

**\E SIGNAL-OR-GROUP-NAME**—return the last expect event. In the case of a group, return a string of the last expect event for each signal of the group. If no expect event has been established, then return ‘~’ (the tilde character).

**\S SIGNAL-OR-GROUP-NAME**—return the last WFC/WFC-list that was established using a substitute (s or S) event in a waveform. In the case of a group, return a string of WFCs for each signal of the group. The actual WFC is determined by the WFCMap statement. If no substitute has been established, then return ‘~’ (the tilde character). For further information, see CompareSubstitute (13.2) and WFCMap statement (10.1).

**\U SIGNAL-OR-GROUP-NAME**—return the last undefined event. In the case of a group, return a string of the last undefined event for each signal of the group. If no undefined event has been established, then return ‘~’ (the tilde character).

**\W SIGNAL-OR-GROUP-OR-SIGNALVARIABLE-OR-WFCCONSTANT-NAME**—return the last WFC/WFC-list that was established using a V or a C statement. In the case of a group, return a string of WFCs for each signal of the group. If the last WFC was established by a parameter using # or %, then the substituted WFC is returned. If no WFC has been established, then return ‘~’ (the tilde character).

Note: This should never happen on the readback of a signal or group because it is required that an initial WFC be defined on the first vector of any pattern.

Read-back functions determine the current value that is defined for a signal, signal group, signal variable, or WFC constant. These read-back functions are identified by the backslash escape character and a function character in front of the name. A space between the escape sequence and the name is optional. Whenever a read-back function is used on the right or left side of a pattern data expression, the backslash escape sequence must be used; i.e., there is no default.

Read-back functions that return events (i.e., \C, \D, \E, \U) are only allowed on signals and groups because the events that make up a WFC are only known when the signal or group is referenced to a waveform table. The read-back functions that return WFCs (i.e., \S, \W) can be used on signals, groups, signal variables, and WFC constants.

When specifying event names on read-back functions, only the single character representation shall be used; i.e., (\C group == HHHLLL) is allowed; (\C signal == H) is allowed; and (\C signal == CompareHigh) is not allowed.

```

677: STIL 1.0 { Design 2005; }
678: Header {
679:   Source "STD 1450.1-2005";
680:   Ann { * sub-clause 15.1 * }
681: }
682:
683: Signals {
684:   SIG Out { WFCMap {S->LH;} }
685:   SIGS[1..4] InOut;
686:   SIG1 In; SIG2 In;
687: }
688: SignalGroups {
689:   GRP = 'SIGS[1..4]';
690: }
691: Variables {
692:   SignalVariable SIGVAR[1..4];
693: }

```

```

694: Timing {
695:   WaveformTable WFT {
696:     Waveforms {
697:       SIG { LHS { '25ns' L/H/S; }}
698:       GRP { LHS { '25ns' L/H/S; }}
699:     }}}
700:
701: Pattern PAT1 {
702:   // Examples of read back functions
703:   W WFT;
704:   V { SIG = L; }
705:   If (\W SIG == L) {} // true since last WFC for this signal was L
706:   V { SIG = S; }
707:   If (\SSIG == H) {} // true if value read by the prior S event mapped to WFC H
708:   If (\DSIG == \eU) {} // true if the last drive event was drive-up
709:   If (\CSIG == \eH) {} // true if the last compare event was compare high
710:   V { GRP = HHHH; }
711:   If (\W GRP == HHHH) {} // true since last WFC for this group was HHHH
712:   V { GRP = SSSS; }
713:   If (\SGRP == HLHL) {} // true if the last group substitute mapped to HLHL
714:   If (\DGRP == \eDDDUUU) {} // true if last drive events for the group were DDDUUU
715:
716:   // Examples of read back into signal variables using read back functions
717:   C {SIGVAR = \W GRP;}
718:   If (\W SIGVAR == HHHH) {}
719:   C {SIGVAR[1] = \W SIG;}
720:   If (\W SIGVAR[1] == H) {}
721: }
722:
723: // The following is an example of an error return on \W function
724: Procedures {
725:   "var_ex" {
726:     C { SIG1 = #; }
727:     If (\W SIG1 == ~) { V { SIG1=B; SIG2=C; } }
728:     Else { V { SIG2=\W SIG1; } } // both SIG1 and SIG2 are assigned to input param
729:   }
730: }
731:
732: Pattern PAT2 {
733:   Call "var_ex"; // no value passed to SIG1
734:   Call "var_ex" {SIG1=A; } // WFC A is passed to SIG1
735: }

```

## 15.2 Vector data mapping and joining—\m, \j

This subclause defines syntax for use within pattern data for either mapping WFCs to other WFCs or for combining WFCs into a single WFC. For example code, see Annex G and Annex H. The syntax is defined as

```

SIGNAL = \m wfc;
GROUP = \m wfc-list;
SIGNAL = \j wfc-1; signal = \j wfc-2;
GROUP = \j wfc-list-1; \j wfc-list-2;

```

where

SIGNAL, GROUP	is the signal that is to be mapped or joined in the pattern data
<i>wfc</i>	is a FROM_WFC that is to be mapped according to a WFCMap statement for a signal
<i>wfc-list</i>	is a string of FROM_WFCs that are to be mapped according to a WFCMap for a signal group
<i>wfc-1, wfc-2</i>	are two FROM_WFC characters that are to be joined according to a WFCMap for a signal
<i>wfclist-1, wfclist-2</i>	are two strings of FROM_WFCs that are to be joined according to a WFCMap for a signal group.

Refer to the WFCMap statement (Clause 10 and Clause 11) for the definition of the mapping/joining syntax. The WFCMap statement, in conjunction with the \m and \j in the pattern data, together specify the resultant waveform character to use in resolving a vector.

The mapping function (\m) allows for a new WFC to be selected for a given WFC in a vector. It is most useful in the case of parameter passing to a macro or procedure that it can be used anywhere a WFC is used. The join function (\j) allows two WaveformCharacters to be specified for the same signal in one vector.

To use the mapping/joining of WFCs, two new flags are added to the cyclized pattern data: \m indicates that the defined mapping from a single WFC should be used; and \j indicates that the defined mapping from two WFCs should be used. When \j is used, both assignments shall be preceded by \j. No more than two \j assignments to the WFC of a given signal may occur in a vector or condition. If there is only one assignment to the WFC of a given signal, then a \j, if present, is ignored.

The \m and \j flags apply to all WFCs up to a space or a semicolon. A space between the \m or \j and the first WFC is optional. The WFCMap is applied only once; i.e., the mapped WFC is not subject to further mapping. If \m or \j are used for a sigref\_expr that does not define a WFCMap, then the components of sigref\_expr are descended until the first corresponding WFCMap is found.

If the vector mapping \m is used but no WFCMap has been defined for the WFC (for sigref\_expr or its components), then the WFC is used unchanged. If the vector mapping \j is used for two WFC assignments, but no WFCMap has been defined for the WFC combination, it is an error condition.

The following examples are of pattern data containing these special mapping characters:

```

736: STIL 1.0 { Design 2005; }
737: Header {
738:   Source "STD 1450.1-2005";
739:   Ann { * sub-clause 15.2 * }
740: }
741: Variables { SignalVariable VARX[1..4]; }
742: Signals {
743:   SIG InOut { WFCMap {AB->1; X->B;} }
744:   SIGS[2..8] InOut;
745: }
746: SignalGroups {
747:   ALL = SIG+SIGS[2..8] { WFCMap {0->L; 1->H;} }
748: }
749: MacroDefs {
750:   MM {
751:     C { VARX = #; }
752:     V { ALL = 1100 \m1100; } // ALL = 1100 HHLL
753:     V { ALL = 1100 \m 1100; } // ALL = 1100 HHLL
754:     V { ALL = \m 1100 1100; } // ALL = HHLL 1100

```

```

755:      V { ALL = 1100 \m####; } //ALL = 1100 HHHH
756:      V { ALL = 1100 \m\W VARX; } //ALL = 1100 LLLL
757:      V { SIG=\jA; SIG=\jB; } //SIG = 1
758:      V { SIG=\jA; SIG=\j\mX; } //SIG = 1
759:      V { SIG=\jA; SIG=\m\jB; } //ERROR - map after join is not allowed
760:      }
761: }
762: Pattern P {
763:   C { ALL=0000 0000; }
764:   Macro MM { SIGS[5..8]=1111; VARX=0000; }
765: }

```

### 15.3 Specifying event data in a pattern—\e

This subclause defines syntax for specifying events in pattern data instead of WFCs. The syntax is defined as

*\e event-list*

The override flag *\e* is added to the *vec\_data* flags for specifying raw event data; an *event-list* is one or more waveform events as defined in Table 13. The *\e* applies to all characters that follow until the occurrence of a space or a semicolon. A space between the *\e* and the first event is optional. The *\e* shall not be used to assign events to signal variables or WFC-constants. See Table 1 for the list of allowed events. For a given signal, there shall be a maximum of one drive event and one compare event defined in any cycle.

**Table 13—Waveform events in pattern data**

	Drive event	Drive action	Compare event	Compare action
High	U	drive high at start of period	H, h	H = high edge strobe at period start; h = start high window compare at period start
Low	D	drive low at start of period	L, l	L = low edge strobe at period start; l = start low window compare at period start
Off	Z	drive off at start of period	T, t	T = tri-state edge strobe at period start; t = start tri-state window compare at period start
Don't compare	n/a	n/a	X, x	do not compare; terminate window compare; takes effect at period start
Unspecified	N	drive high or low (application can choose)	S, s	S = compare H/L/T at period start; s = start window compare h/l/t at period start; actual state unknown; also used for block read
Unknown	?	input/output and level are unknown	?	input/output and level are unknown

An example of when the event data are useful is in simulation. When specifying data for simulation stimulus, it is often the case that the desired level of a given signal (input drive or output compare) is known, whereas the exact timing waveform that is appropriate for the cycle is not known. It, in many cases, is why simulation for test generation is performed. The simulator (or STIL consuming cosimulation engine) would use the raw data and the conditions of the device under simulation to determine the WaveForm Character, which will be included in the test pattern.

The following example is of a Pattern block indicates how data can be specified without specifying the waveform characters to be used to reference to the WaveformTable. A WaveformTable reference is still made in a raw pattern, because the successful waveform will ultimately come from that table. This example also illustrates the ability to mix raw and resolved data (see the CLK signal):

```

766: STIL 1.0 { Design 2005; }
767: Header {
768:   Source "STD 1450.1-2005";
769:   Ann { * sub-clause 15.3 * }
770: }
771:
772: Signals {
773:   CLK In; SIGA InOut; SIGB In;
774:   BUSX[1..8] In;
775:   BUSY[1..8] Out;
776: }
777:
778: Timing {
779:   WaveformTable MY_TIMING {
780:     Waveforms {
781:       'CLK+SIGA+SIGB+BUSX+BUSY' {
782:         A { /* waveform defs for A */ }
783:         B { /* waveform defs for B */ }
784:       }
785:     }
786:   }
787: }
788:
789: Pattern WITH_RAW_EVENTS {
790:   WaveformTable MY_TIMING;
791:   V { CLK = 1; SIGA = \eU; SIGB = \eL; }
792:   V { SIGA = \eH; SIGB = \eD; }
793:   V { BUSX = \eUUUUDDDD; }
794:   V { BUSY = AABB \eXXX A; }
795: }

```

## 15.4 Using expressions within pattern data

When expressions are used in pattern data, they shall always be enclosed in parentheses. The only place where expressions are allowed in pattern data is to represent the integer value for the operators: \d, \l, \r. The following examples are of pattern data using expressions:

```

V { FOO = XXX \l(K+1) XXXXX XXXXX; }
V { FOO = XXX \r(K+1) X; }
V { FOO = \d (K+1); }
V { FOO = XXX \l(LEN) \dLH (K+1) XXX; }

```

## 16. Pattern statements

This clause defines extensions to STIL.0, Clause 22.

This clause defines additional statements that are allowed within the Pattern block. All constructs and definitions of STIL.0, Clause 22 remain in effect.

### 16.1 Additional Pattern syntax

#### Pattern statements:

- ( LABEL : ) **If** *boolean\_expr* { ( PATTERN\_STATEMENTS ) \* } ( **Else** { ( PATTERN\_STATEMENTS ) \* } ) (1)
- ( LABEL : ) **While** *boolean\_expr* { ( PATTERN\_STATEMENTS ) \* } (2)
- ( LABEL : ) **F(ixed)** { (*cyclized-data*) \* (*non-cyclized-data*) \* } (3)
- ( LABEL : ) **E(quivalent)** ( ( \m ) *sigref\_expr* ) \* ; (4)
- ( LABEL : ) **LoopData** { ( PATTERN\_STATEMENTS ) \* } (5)
- ( LABEL : ) **Loop** *integer\_expr* { ( PATTERN\_STATEMENTS ) \* } (6)
- ( LABEL : ) **ActiveScanChains** ( GROUP\_OR\_CHAIN\_NAME ) + ; (7)
- ( LABEL : ) **AllowInterleave** *sigref\_expr* ; (8)
- ( LABEL : ) **BreakPoint** { } (9)
- ( LABEL : ) **X** X\_REF\_ID ; (10)

- (1) **If/Else** *boolean\_expr*: Defines a block of PATTERN\_STATEMENTS to be executed only if the *boolean\_expr* is true. The value of *boolean\_expr* is determined at the point that execution reaches this statement. The optional **Else** block contains pattern statement that are executed if *boolean\_expr* is false. For example code, see Annex D.

The following rules and restrictions apply to If/Else statements:

- The If/Else shall not be used to condition the execution of a Shift block (see STIL.0, Subclause 24.5 for the definition of Shift operation and data substitution).
- The statements within an If/Else block have the same effect and scope as if they were to be executed outside of the block; i.e., the statement “If (xxx) { Fixed yyy; }” will be in effect for the rest of the pattern if ‘xxx’ evaluates to True.

- (2) **While** *boolean\_expr*: Defines a block of PATTERN\_STATEMENTS to be repeatedly executed as long as the *boolean\_expr* is *True*. The value of *boolean\_expr* is reevaluated each time execution reaches this statement. For example code, see Annex E.

The rules and restriction defined for the If/Else statements also apply to the While statement (see If/Else statements).

- (3) **F(ixed)**: This statement allows the specification of signals that are to remain in one state for the remainder of the pattern or procedure. The signals may define a fixed static state or may be assigned a WFC that is to be repeated. The list of signals in the *sigref\_expr* shall not be used in vector, condition, macro, or procedure statements for the remainder of the pattern. If a signal is fixed to a WFC, then that WFC shall be defined within each WFT that is selected for each vector. If a static fixed value is desired, then this can be accomplished using the \e syntax (see 15.3), which requires no WFC/WFT definitions. For example code, see 12.3, 16.2, and Annex J.
- (4) **E(quivalent)**: Statement to establish WaveformCharacter relationships between signals. Equivalent statements do not define event sequences and are executed in zero-time, in the same manner as Condition statements defined in STIL.0. See 16.2 for a complete definition of this statement. For example code, see 16.2 and Annex J.
- (5) **LoopData**: The Loop construct is expanded to support the keyword Data. This construct is used only for Loops inside Macro and Procedure bodies. This Loop statement iterates, replacing each ‘#’ parameter with a data value until no more data values are defined. Short parameter strings are extended by replicating the last WFC until the longest parameter has been exhausted. It is an error if all data values are not consumed by the ‘#’ parameters in the Loop. Early termination may be done by use of “If ( ) GoTo LABEL;”. For example code, see 16.3 and Annex E.

- (6) **Loop *integer\_expr*:** The Loop construct is extended to allow a variable expression of type integer. The Loop statement iterates for the number of times indicated by the evaluation of the expression. The Loop value may be either an expression or a constant, and in either case the loop value is evaluated only once at the beginning of the loop. Changing of a variable within a loop has no effect on the current loop count. Note: For run time control of a loop, use the While statement. For example code, see 16.4.
- (7) **ActiveScanChains:** This statement is used to specify the scan chains that are active for all vectors and shift operations that follow, up to the next ActiveScanChain statement or the end of the current Pattern or Procedure. This statement (like the Shift statement) shall only be used within a Procedure or a Macro block. Multiple ActiveScanChains statements may occur within a Macro or Procedure block. This statement has no effect on the execution of the pattern on an ATE system, but it is used by design tools to verify correct loading/unloading of scan chains in a pattern. For example code, see 14.3 and 14.5).
- (8) **AllowInterleave:** The AllowInterLeave statement identifies a set of signals in a Pattern, Procedure, or Macro that are shared between multiple patterns running in LockStep. Any signal identified as an interleaved signal shall not maintain the last defined WFC on that signal (no implicit persistence), but each resolved Vector shall define a WFC for that signal. For example code, see Annex F. The following rules apply to interleaved signals:
- The AllowInterleave is only allowed in conjunction with ParallelPatList block with LockStep.
  - The AllowInterleave, when present, shall appear before the first Vector of the context (Pattern, Procedure, or Macro)
  - For each vector, only one pattern shall define a WFC for an interleaved signal.
  - On a given vector, it is allowed for multiple patterns to define a WFC on an interleaved signal as long as it is the same WFC. Otherwise, it is an error. This rule is true regardless of whether the AllowInterleave has been specified (i.e., the base pattern may have common signals with common activity, and when a procedure or macro is invoked, the signals go into interleave mode for the duration of the procedure or macro).
  - The WFC of an interleaved signal is not carried over to the next Vector in sequence.
  - If, on a given vector, no LockStep pattern defines a WFC, then it is an error.
  - When a procedure or macro is invoked, the interleaved signals do NOT persist into or out of the procedure or macro. The interleaving is defined only for the duration of the procedure or macro in which it is specified.
- (9) **BreakPoint:** The BreakPoint functionality is as defined in STIL.0, with additional behaviors to support parallel operation of patterns under a ParallelPatList block. A BreakPoint block can be repeated as necessary to align parallel patterns. The definition of the application of this capability is to be specified in the standard or tool that uses it. See 12.3 for applications in a PatternBurst.
- (10) **X:** The X statement (which is short for cross-reference) is used to identify key places in the pattern that are used for collecting data during the device test process and reporting that data to a post-processing tool. The post-processing tool, most likely, is closely related to the tool that created the STIL pattern in the first place. The X statement is optional and may be used in any pattern, macro, or procedure. Refer to PatternFailReport (Clause 20) for definition of usage of the X statement.

**X\_REF\_ID:** The keyword is followed by an identifier that may be either a user-defined name (according to the rules of STIL.0, Subclause 6.8) or an integer.

If the X statement is present in a Pattern block, it typically indicates the beginning of a “pattern unit” (where a “pattern unit” is a sequence of vectors that makes up a test as created by an automatic test pattern generation tool), which is typically used in the context of scan and comprising a scan-in, some activity to DUT pins, followed by a scan-out.

If the X statement is present in a Macro or Procedure block, then the associated identifier or integer is appended to the identifier from the calling pattern according to rules as defined in the PatternFailReport (Clause 20).

## 16.2 Vector data constraints—F, E

Structured test patterns often have signals constrained to have a certain value or waveform during a pattern sequence. It may be required, for example, for ATPG scan rules checking (such as a test mode signal always active) or for differential scan or clock inputs. These constructs help reduce pattern volume, as the value of a constraint signal does not need to be specified explicitly in the pattern data. Also, ATPG rules checking requires signal constraint information as input.

Two new STIL pattern statements are defined:

```
( LABEL : ) F(ixed) { (cyclized-data)* }
```

The Fixed statement defines stimulus and/or response fixed for all subsequent vectors, until the end of the current Pattern or Procedure. Every Procedure starts its own environment, not inheriting Fixed data from its caller. After return from a procedure, the Fixed values of the caller are reinstated. Subsequent vectors may redundantly define the same WaveformCharacter assignments as in the Fixed statements; however, no vector is allowed to define a WaveformCharacter assignment in contradiction with the Fixed statements in effect. The Fixed pattern statement does not result in a tester cycle, but constrains and sets values for all following cycles within its scope:

```
( LABEL : ) E(quivalent) ( (\m) sigref_expr )* ;
```

The Equivalent statement defines an equivalence relationship between two or more signals. Equivalence means that the signals are assigned the same WaveformCharacter. Or a signal or signal expression is optionally preceded by `\m` to indicate that the WaveformCharacter is to be mapped before the equivalence is applied. All signal expressions in each `sigref_expr` shall have the same number of signals; the first signals of the signal expression are equivalent, the second signals of all signal expressions are also equivalent, and so on. Note that even though the equivalence can be specified on a per-pattern or per-procedure basis, the mapping function is per-signal and cannot be redefined.

The scope of the equivalence extends to all subsequent vectors until the end of the current Pattern or Procedure. Subsequent vectors may redundantly define the same WaveformCharacter assignments as in the equivalent statements; however, no vector is allowed to define a WaveformCharacter assignment in contradiction with the equivalent statements in effect. Every Procedure starts its own environment, not inheriting Equivalent data from its caller. After return from a procedure, the Equivalent values of the caller are reinstated.

Vectors may assign WaveformCharacters to any nonempty subset of equivalent signals and the equivalence relationship extends the assignment to the other signals in the equivalence class. At least one signal in each equivalence class must be unmapped and must have WFC assignments in the pattern or procedure.

The Equivalent pattern statement does not result in a tester cycle, but it constrains and sets values for all following cycles within its scope.

The following examples illustrate usage of Fixed and Equivalent statements:

```
796: STIL 1.0 { Design 2005; }
797: Header {
798:   Source "STD 1450.1-2005";
799:   Ann { * sub-clause 16.2 * }
800: }
801:
802: Signals {
803:   SIG_NAME1 In; SIG_NAME2 In; SIG_NAME3 In; SIG_NAME4 In;
804:   A[15..0] InOut; B_LOW[7..0] InOut; B_HIGH[15..8] InOut;
805: } // end Signals
```

```
806:
807: SignalGroups {
808:   A = 'A[15..0]';
809:   B = 'B_HIGH[15..8]+B_LOW[7..0]' {
810:     WFCMap {
811:       0->1; 1->0; Z->Z; N->N; L->H; H->L; T->T; X->X;
812:     } // end WFCMap
813:   } // end B
814: } // end SignalGroups
815:
816: /* SIG_NAME1 and SIG_NAME2 have the same WaveformCharacter;
assigning a WaveformCharacter to one of these signals is equivalent to
assigning values to both signals. */
817: Pattern PAT1 {
818:   Equivalent SIG_NAME1 SIG_NAME2;
819:   Fixed { SIG_NAME3 = P; SIG_NAME4 = \eD; }
820: } // end Pattern
821:
822: /* A more complex example is of two buses of bidirectional signals
that are complementary both as inputs and as outputs: */
823: Pattern PAT2 {
824:   Equivalent A \m B;
825:   V { A[15..0]=0; B[8]=H; } // implies also: B_LOW[0]=1; A[8]=L;
826: }
```

### 16.3 Shift and LoopData statements

The Shift and LoopData statements perform similar functions; they loop over a sequence of pattern statements; they may be used only in procedures or macros; they take parameters from the calling pattern statements and apply the parameters to signals; they take signal-variable data and apply it to signals. The difference between the Shift and LoopData is that the Shift is specifically designed in support of scan load/unload operations, whereas LoopData is a general construct for providing data-dependent loops. The following rules further identify the differences.

#### 16.3.1 Rules for Shift

- a) Within a Shift block, the parameters to be shifted shall be determined by the presence of a ScanIn or ScanOut attribute in the Signal or SignalGroups block. It is as opposed to implying the shift from the presence of # characters.
- b) For each ScanIn/ScanOut signal within a Shift block, the shift data can be defined in three ways: (1) a parameter referenced by the #% notation; (2) a parameter defined by a signal-variable; or (3) an in-line series of WFCs.
- c) The loop size for a Shift block is determined by the length of the longest normalized data set assigned to scan signals in the Shift block (adjusted for pre-Shift and post-Shift consumption).
- d) Padding rules (i.e., data to be supplied when a parameter is less than the length required by the longest parameter) are in accordance with STIL.0, Subclause 24.5.
- e) If there are multiple Shift blocks in a macro or procedure, a given scan-signal shall appear in only one Shift block.
- f) Parameters to scan-signals shall not be used within any If/Else/While construct. It applies to statements within a Shift block, pre/post-shift statements, and the Shift statement.

- g) Within a Shift block, only scan signals shall have parameterized data assignments. All non-scan signals shall be assigned constant values within the Shift block or retain the a constant value from the last assignment before the Shift block.

### 16.3.2 Rules for LoopData

- a) For each signal within a LoopData block, the parameterized data shall be defined in one of three ways: (1) a parameters defined by #% notation; 2) parameters defined by a signal-variable; or (3) an in-line series of WFCs.
- b) Within a LoopData construct, at least one waveform character assignment shall contain a list of parmeterized data; else the loop will execute only once.
- c) Looping shall continue until all parameters have been exhausted; i.e., the loop size is determined by the length of the longest parameter.
- d) Short parameter lists shall be extended by replicating the last WFC as necessary.
- e) Scan-signals (should they appear inside a LoopData block) are processed as non-scan-signals (i.e., scan padding rules do not apply). Note: If a Shift follows a LoopData, then there shall be no data left to shift.
- f) The conditional statements, If/Else/While, shall be allowed within the LoopData, on pre-LoopData statements that consume parameters and on a LoopData block.
- g) If parameter data are consumed before a LoopData block, then the LoopData shall loop according to the remaining data.

The following example is of LoopData and Shift:

```

827: STIL 1.0 { Design 2005; }
828: Header {
829:   Source "STD 1450.1-2005";
830:   Ann { * sub-clause 16.3 * }
831: }
832:
833: Signals {
834:   SIG1 In;
835:   SI1 In { ScanIn 14; }
836:   SI2 In { ScanIn 17; }
837:   SI3 In { ScanIn 11; }
838:   SI4 In { ScanIn 20; }
839: } // end Signals
840: Variables {
841:   SignalVariable ABC[1..4] ;
842:   SignalVariable DEF[1..3] ;
843:   SignalVariable XYZ[1..20] ;
844: } // end Variables
845:
846: Procedures ALL {
847:   VARIABLE_LOOP {
848:     LoopData { V { SIG1=#; }}
849:   }
850:   SHIFT_LOAD {
851:     C { 'SI1+SI2+SI3+SI4+SI5+SI6' = 000000; }
852:     C { ABC = #; DEF = #; XYZ = #; }
853:     Shift {
854:       V {
855:         SI1 = 10101 \W ABC 10101;           // ABC is a SignalVariable

```

```

856:      SI2 = 10101 \W ABC \W DEF 10101;      //ABD, DEF are two SignalVariables
857:      SI3 = 10101 \W XYZ[1] 10101;          // with indexed SignalVariable
858:      SI4 = 10101 \W XYZ[11..20] 10101;      // with multi-bits of a SignalVariable
859:      } //end V
860:    } //end Shift
861:  } //end SHIFT_LOAD
862: } //end Procedures
863:
864: Pattern RUN1 {
865:   Call VARIABLE_LOOP { SIG1=010; }           //Loop 3 times ('010' is WFC-list)
866:   Call VARIABLE_LOOP { SIG1=00000; }        //Loop 5 times
867: } //end Pattern
868:
869: Pattern RUN2 {
870:   Call SHIFT_LOAD {
871:     ABC=1100;
872:     XYZ[1..20]=11111000001111100000;
873:     DEF=101;
874:   } //end Call
875: } //end Pattern

```

## 16.4 Loop statement using an integer expression

The Loop statement as defined in STIL0 is extended to allow the use of integer expressions. This extension applies to STIL.0, Subclause 22.6 (Loop statement) and to STIL.0, Subclause 22.7 (MatchLoop statement). The following examples are of this usage:

```

876: STIL 1.0 { Design 2005; }
877: Header {
878:   Source "STD 1450.1-2005";
879:   Ann {* sub-clause 16.4 *}
880: }
881: Variables {
882:   Integer COUNT { InitialValue 100; }
883: }
884: Signals { CLK In; A[1..8] In; B[1..8] Out; }
885: SignalGroups {
886:   SOME_PINS = 'A[1..8]+B[1..8]';
887: } //end SignalGroups
888: Procedures {
889:   PROC1 {
890:     C { COUNT = #; CLK=0; SOME_PINS=\r8 0 \r8 X; }
891:     Loop COUNT { V { CLK = P; } }
892:     V { SOME_PINS = 11110000 HHHHLLLL; }
893:   }
894: }
895: Pattern PAT1 {
896:   Call PROC1 { COUNT := 200; }
897: }

```

## 16.5 MergedScan function

The MergedScan() function returns an integer value that is used to determine if scan operations are to be applied in a merged or unmerged fashion. The behavior of merged and unmerged scan is described in STIL.0, Subclause 5.5. The evaluation of this function shall be in the context of the STIL consumer environment. If the STIL consumer environment has no specific needs for altering the merged scan operation, then this function shall return a default value of 1 (i.e., boolean true).

This construct in conjunction with the If and Else constructs allow the STIL producer to generate STIL patterns/procedures/macros that represent both merged and unmerged scan operations. See Annex M for an example of syntax and usage.

## 17. Procedure and macro data substitution

This clause defines extensions to STIL.0, Subclause 24.5.

All constructs and definitions of STIL.0, Subclause 24.5 remain in effect, with the following additional capabilities.

### 17.1 Nested procedure and macro cells

To pass data through nested procedure and macro calls the following restrictions apply:

- Data passed from one procedure or macro, into another procedure or macro, shall use the same argument name to reference that data, as was used to specify that data to that procedure. Data cannot be passed into another procedure or macro call using a different signal or group name on that data than was used to pass that data into the procedure.
- Data passed into another procedure or macro is considered to consume all data values passed into the procedure, in that procedure or macro call. The procedure or macro calling another procedure or macro shall not reference the parameters passed to that called procedure or macro directly, except to pass them to the called procedure or macro.

An example of a proper environment to pass data through procedure or macro calls is as follows:

```

898: STIL 1.0 { Design 2005; }
899: Header {
900:   Source "STD 1450.1-2005";
901:   Ann {* sub-clause 17.1 *}
902: }
903:
904: Signals {
905:   SCANSIG In { ScanIn; }
906:   INS[1..5] In;
907:   CLK In;
908:   SCANMODE In;
909: }
910:
911: SignalGroups {
912:   _INS = 'INS[1..5];
913: }
914:
915: Procedures {
916:   DO_SHIFT {

```

```

917:     Shift { V { SCANSIG = #; CLK = P; }}
918:   }
919:   SETUP_AND_SHIFT {
920:     C { _INS = #####; }
921:     V { CLK = 0; SCANMODE = 1; }
922:     Call DO_SHIFT { SCANSIG = #; }
923:   }
924: } //end Procedures
925:
926: Pattern "RUN_SHIFTS" {
927:   Call SETUP_AND_SHIFT { _INS = 00101; SCANSIG = 010001010100; }
928: }

```

## 17.2 Passing parameters to variables

Signal variables and integer variables operate in the same manner as signals and groups when used as formal parameters are passed to procedures and macros (for definition of formal parameters, see 5.12). The # and % operators are used to transfer the parameter to the named signal, group, or variable within a procedure or macro. As is shown in the following example, multiple data values can be passed as parameters to a macro or procedure:

```

929: STIL 1.0 { Design 2005; }
930: Header {
931:   Source "STD 1450.1-2005";
932:   Ann { * sub-clause 17.2 * }
933: }
934:
935: Signals {
936:   X[3..0] In;
937:   Y[7..0] Out;
938: }
939: SignalGroups {
940:   Y = 'Y[7..0]' { Base Hex AB; }
941: }
942: Variables {
943:   SignalVariable SV[3..0];
944:   Integer INT;
945: }
946: Macrodefs {
947:   M {
948:     C { SV = #; INT := #; } // get 1st parameter
949:     V { X = \W SV; Y = \d INT; } // X=0000; Y=AAAA ABAB;
950:     C { SV = #; INT := #; } // get 2nd parameter
951:     V { X = \W SV; Y = \d INT; } // X=0001; Y=AAAB BBBB;
952:     C { SV = #; INT := #; } // get 3rd parameter
953:     V { X = \W SV; Y = \d INT; } // X=0100; Y=BBBB BBBB;
954:   }
955: }
956: Pattern P {
957:   C { X = 0000; Y = XXXX XXXX; }
958:   Macro M {
959:     SV = 0001 0010 0100;
960:     INT = 5 119 0xFF;
961:   }
962: }

```

### 17.3 Default value of formal parameters

Signals, groups, and variables when specified within macro or call statements are formal parameters to the macro or procedure. The value(s) specified take effect when the signal, group, or variable is set to # or %. The Condition statement is used to define the default state within a macro or procedure as shown in the following example. If not specified in a macro or procedure, the default value for variables is as defined by its InitialValue statement; the default state for signals and groups is as defined by its DefaultState statement:

```

963: STIL 1.0 { Design 2005; }
964: Header {
965:   Source "STD 1450.1-2005";
966:   Ann {* sub-clause 17.3 *}
967: }
968:
969: Signals {
970:   SIG In;
971: }
972: Variables {
973:   SignalVariable SIGV[1..4];
974:   Integer INT;
975: }
976: MacroDefs {
977:   MAC {
978:     C { SIG=0; SIGV=XXXX; INT:=0; }
979:     V {                               // 1st, 2nd, 3rd usage of MAC
980:       SIG=#;                          // 0, XXXX, 0
981:       SIGV=#;                          // 1, 1001, 5
982:       INT:=#;                          // 0, XXXX, 0
983:     }
984:   }
985: }
986: Pattern PAT {
987:   Macro MAC { }
988:   Macro MAC { SIG1=1; SIGV=1001; INT:=5; }
989:   Macro MAC { }
990: }

```

### 17.4 Data substitution using WFCConstant and SignalVariable

Two new data types are available for doing data substitution in macros and procedure: WFCConstant and SignalVariable. Refer to Clause 9 for the syntax definition for these statements.

A WFCConstant is used in the same way that an explicit string of WFC characters is used in STIL.0. Consider the following example where SI1 and SI2 are provided identical data strings; only SI2 is defined using a defined constant.

A SignalVariable is used in a similar way to the # operator, which allows variable data to be substituted onto a signal in a macro or procedure. The difference being that the data that are passed as a formal parameter in a macro/procedure call are not locked to the named signal or group. This process allows more freedom in the way that the data are consumed. Consider the following example, where the data passed into two similar procedures are identical. In the case of SOUT1, the data have to be consumed by a signal named SOUT1, whereas in the case of SOUT2, the data have been split onto two signals:

```
991: STIL 1.0 { Design 2005; }
992: Header {
993:   Source "STD 1450.1-2005";
994:   Ann {* sub-clause 17.4 *}
995: }
996:
997: Signals {
998:   SI1 In { ScanIn; }
999:   SI2 In { ScanIn; }
1000:  SO1 Out { ScanOut; }
1001:  SO2 Out { ScanOut; }
1002:  SO3 Out { ScanOut; }
1003: }
1004: Variables {
1005:   WFCConstant FIXED = 1100;
1006:   SignalVariable SV[0..7];
1007: }
1008: Procedures {
1009:   SIN {
1010:     Shift {
1011:       // the data on SI1 and SI2 are identical
1012:       V { SI1 = 1100####; SI2 = \WFIXED ####; }
1013:     }
1014:   }
1015:   SOUT1 {
1016:     Shift {
1017:       V { SO1 = #; } // data SO1=HHHHLLLL
1018:     }
1019:   }
1020:   SOUT2 {
1021:     C { SV = #; }
1022:     Shift {
1023:       V { SO2 = SV[0..3]; SO3 = SV[4..7]; } // data SO2=HHHH, S3=LLLL
1024:     }
1025:   }
1026: }
1027: Pattern PAT {
1028:   Call SIN { SI1 = 0101; SI2 = 0101; }
1029:   Call SOUT1 { SO1 = HHHHLLLL; }
1030:   Call SOUT2 { SV = HHHHLLLL; }
1031: }
```

## 18. Environment block

The Environment block contains constructs to cross-reference STIL information with other environments, for example, a simulation or layout netlist environment. The Environment block may be used by tools that operate in the specified Environment and access STIL data. The Environment block may also contain additional information specific for the particular environment.

Because the Environment block references information not contained in the STIL environment, no STIL blocks reference the Environment block. The Environment block is self-contained, but internally it may reference other STIL blocks as necessary to associate STIL information with the external context.

The Environment block may contain additional blocks that have not been defined as part of the STIL.1 standard and that do not need to be declared (with a UserKeywords statement) before they appear in the Environment block. These undeclared blocks are allowed with the same restrictions as a UserKeyword block. It is expected that the definition of these undeclared blocks is found in association with an extension definition. See Clause 7 for extension reference constructs. These blocks, when present, shall follow all STIL syntax requirements for UserKeyword blocks. These blocks contain information specific to an Environment context (in STIL syntax form).

### 18.1 Environment syntax

```

Environment (ENV_NAME) { (1)
  (InheritEnvironment ENV_NAME ;)* (2)
  (NameMaps (MAP_NAME) { (3)
    (InheritNameMap (ENV_NAME::) (MAP_NAME) ;)* (4)
    (InheritNameMap (ENV_NAME::) (MAP_NAME) { (5)
      (Prefix "PREFIX_STRING";) (5)
      (Separator "SEPARATOR_STRING";) (6)
      (ScanCells { (SCAN_CELL_NAME ;)* })* (7)
    })* // end InheritNameMap
    (Prefix "PREFIX_STRING";) (8)
    (Separator "SEPARATOR_STRING";) (9)
    (ScanCells { ((SCAN_CELL_NAME) "MAP_STRING";)* })* (10)
    (Signals { (SIG_NAME "MAP_STRING";)* })* (11)
    (SignalGroups (DOMAIN_NAME) { (GROUP_NAME "MAP_STRING";)* })* (12)
    (Variables { (VAR_NAME "MAP_STRING";)* })* (13)
    (AllNames { (ANY_NAME "MAP_STRING";)* })* (14)
  })* // end NameMaps
  (FileReference "FILE_PATH_NAME";)* (15)
  (FileReference "FILE_PATH_NAME" { (16)
    Type FILE_TYPE ; (16)
    Format FILE_FORMAT ; (17)
    Version "VERSION_NUMBER" ; (18)
  })* // end FileReference
  (ENVIRONMENT_DEFINED_STATEMENTS)* (19)
} // end Environment

```

- (1) **Environment:** Defines an Environment block.

ENV\_NAME: Optional name of the Environment block. A single global Environment block may be defined without a name.

- (2) **InheritEnvironment:** Statement to reference a previously defined Environment block, to incorporate the definitions of that block as part of the current block. Only statements with defined keywords are inherited, that is, SignalGroups, NameMaps, and FileReference statements or blocks. See the InheritNameMap statement for information on how NameMaps are inherited. If a FileReference block has the same name locally as an inherited FileReference, the local definition will be used. ENVIRONMENT\_DEFINED\_STATEMENTS are not inherited unless the inheritance rules are specified by some other STIL extension and that extension is identified in the file header. An un-named

Environment block is not automatically inherited and cannot be explicitly inherited (because it has no reference name).

- (3) **NameMaps:** Defines one or more blocks containing references of STIL names to names defined in an external Environment. The MAP\_NAME is optional when a single NameMaps block is present; the MAP\_NAME is required if subsequent NameMaps blocks are present. The MAP\_NAME shall be unique across all NameMaps blocks in a single environment when present.
- (4) **InheritNameMap:** Statement to reference a previously defined NameMaps block inside an Environment block, to incorporate the definitions of the NameMaps section of that block as part of the current block.

If a NameMaps block in the current environment is being referenced, only the name of that block is required. If a NameMaps block in another environment block is being referenced, then the ENV\_NAME:: is required, followed by the name of the desired block. If the NameMaps block to be inherited is the global block, then the MAP\_NAME is not required. A local definition of a SCAN\_CELL\_NAME, SIGNAL\_NAME, GROUP\_NAME, VAR\_NAME, or ANY NAME will override any inherited definitions for those names.

An alternative form of InheritNameMap uses the { } format and allows for additional capabilities relative to the scan-cells information.

- (5) **Prefix:** This statement, when used within the context of InheritNameMap, causes the PREFIX\_STRING to be prepended to each MAP\_STRING definition that is inherited.
- (6) **Separator:** This statement, when used within the context of InheritNameMap, causes the SEPARATOR\_STRING to be used on each MAP\_STRING definition that is inherited.
- (7) **ScanCells:** This statement, when used within the context of InheritNameMap, is used to define a list of SCAN\_CELL\_NAMES. This list of names is to be correlated to a list of MAP\_STRINGS in the inherited block and along with the prefix and separator from the mapped name definitions for each cell. It is an error condition if the number of MAP\_STRINGS is less than the number of SCAN\_CELL\_NAMES. If the number of MAP\_STRINGS is greater than the number of SCAN\_CELL\_NAMES, the additional MAP\_STRINGS are ignored. Note that the correlation between cell and map strings is by position in the list.
- (8) **Prefix:** This statement, when used within the context of NameMaps, causes the PREFIX\_STRING to be prepended to each MAP\_STRING definition that occurs within the current NameMaps block.
- (9) **Separator:** This statement, when used within the context of NameMaps, causes the SEPARATOR\_STRING to be used on each MAP\_STRING definition that occurs within the current NameMaps block.
- (10) **ScanCells:** This statement, when used within the context of NameMaps, can take two forms: one with the SCAN\_CELL\_NAME defined and one without.

In the case in which the SCAN\_CELL\_NAME is defined, a direct mapping of the name of the cell is made to the defined MAP\_STRING. The MAP\_STRING parameter is required.

In the case in which only one parameter is defined, then it is interpreted as the MAP\_STRING. It is the form to be used when the list of MAP\_STRINGS is to be inherited.

- (11) **Signals:** Required if SIG\_NAME map information is provided.  
SIG\_NAME "MAP\_STRING": Statement to map the STIL SIG\_NAME defined in a Signals block, into the name used in the external Environment.
- (12) **SignalGroups:** Required if GROUP\_NAME map information is provided.  
GROUP\_NAME "MAP\_STRING": Statement to map the STIL GROUP\_NAME defined in a SignalGroups block, into the name used in the external Environment.
- (13) **Variables:** Required if VAR\_NAME map information is provided.  
VAR\_NAME "MAP\_STRING": Statement to map the STIL VAR\_NAME defined in a Category or Spec block, into the name used in the external Environment.

- (14) **AllNames**: This optional block supports an Environment that does not partition name spaces with the same structure as STIL and supports mapping of names from additional STIL constructs (such as objects in user-defined blocks).
- ANY\_NAME “MAP\_STRING”: Statement to map any STIL NAME, into the name used in the external Environment.
- (15) **FileReference** “FILE\_PATH\_NAME”: The FileReference statement is used within an Environment block to specify various other files associated information. The content and application of the referenced files is not specified in STIL, but it is the responsibility of the tool or application using the Environment block. For instance, if the patterns are in STIL, then they would be Included, but if the patterns were in WGL or some tester-specific format, this mechanism would be used to reference that data.
- (16) **Type** FILE\_TYPE: Specifies the type of this file. The allowed file types are not defined as part of this standard.
- (17) **Format** FILE\_FORMAT: Specifies the format of this file type. The allowed file types are not defined as part of this standard.
- (18) **Version** “VERSION\_NUMBER”: A quoted string identifying the version of this file. The format and information of the VERSION\_NUMBER is dependent on the file type and format, and it is not defined here.
- (19) ENVIRONMENT\_DEFINED\_STATEMENTS: These are semicolon-terminated or braced statements that follow the STIL requirements but do not require a UserKeyword statement to identify these blocks. The environment-defined statement and semantics associate with the specification for the extension as identified in the STIL {} block.

## 18.2 MAP\_STRING syntax

The target MAP\_STRING maps STIL names into an external environment. If the target name contains either a double quote or a backslash, then the following syntax shall be used in the target string. It is an error for any character other than double quote or backslash to follow a backslash character.

double-quote: “blah\_blah”\“blah\_blah”

backslash: “blah\_blah\\blah\_blah”

error: “blah\_blah\blah\_blah”

A typical usage of this backslash notation is in mapping strings into Verilog names. The following example is of this transformation:

MAP\_STRING in STIL: “\\”abc[15]”

Resultant string : \”abc[15]

NOTE—Per STIL.0 definition, double quotes are only delimiters and are not allowed as part of the string.

## 18.3 NameMaps example

The NameMaps block relates STIL names to external environments. An example of an application of this construct, for a Verilog-style environment, is shown in Figure 4.

```
module top_test;// Testbench Module
    integer pattern;
    wire [0:2] PO; // Primary Outputs Vector
    reg [0:3] PI;// Primary Inputs Vector
    wire A, B1, C1, D11, Q11, Q21;// Signals
    assign A = PI[0], B1 = PI[1], C1 = PI[2], D11 = PI[3];
    assign PO[0] = Q11, PO[1] = Q21;
        // Design Instance
    top dut (.A(A),.B1(B1),.C1(C1),.D11(D11),.Q11(Q11),.Q21(Q21));
endmodule
```

**Figure 4—Verilog test bench example for NameMaps**

```
1032: STIL 1.0 { Design 2005; }
1033: Header {
1034:   Source "STD 1450.1-2005";
1035:   Ann {* sub-clause 18.3 *}
1036: }
1037:
1038: Signals { "A" In; "B1" In; "C1" In; "D11" In; }
1039: SignalGroups { _PI = ' "A" + "B1" + "C1" + "D11" ' ; }
1040:
1041: Environment MY_VERILOG_TESTBENCH {
1042:   NameMaps VECTOR_ASSOCIATIONS {
1043:     Signals {
1044:       "A" "top_test.PI[0]";
1045:       "B1" "top_test.PI[1]";
1046:       "C1" "top_test.PI[2]";
1047:       "D11" "top_test.PI[3]";
1048:     }
1049:     SignalGroups {
1050:       _PI "top_test.PI";
1051:       _PO "top_test.PO";
1052:     }
1053:     Variable { _PATCOUNT "PATTERN"; }
1054:   } // end NameMaps
1055:
1056:   NameMaps WIRE_ASSOCIATIONS {
1057:     Signals {
1058:       "A" "top_test.A";
1059:       "B1" "top_test.B1";
1060:       "C1" "top_test.C1";
1061:       "D11" "top_test.D11";
1062:     }
1063:     SignalGroups {
1064:       _PI "top_test.PI";
1065:       _PO "top_test.PO";
1066:     }
1067:     Variable { _PATCOUNT "PATTERN"; }
1068:   } // end NameMaps
1069: } // end Environment
```

**18.4 Compact scan-cell mapping using InheritNameMap**

```

1070: STIL 1.0 { Design 2005; }
1071: Header {
1072:   Source "STD 1450.1-2005";
1073:   Ann {* sub-clause 18.4 *}
1074: }
1075: Environment HIERARCHICAL {
1076:   // A hierarchical NameMap for scan cells
1077:   // See the FLAT example below to see the expanded net names
1078:   NameMaps C {
1079:     ScanCells {"CELL1"; "CELL2";} // Only the MAP_STRING portion of the map
1080:   }
1081:   NameMaps D {
1082:     InheritNameMap C {
1083:       Prefix "C3"; // String to prepend to MAP_STRINGS in inherited NameMap
1084:       Separator "/";
1085:     }
1086:     ScanCells {"CELL3";}
1087:   }
1088:   NameMaps TOP {
1089:     Separator "/";
1090:     InheritNameMap C {
1091:       Prefix "C1";
1092:       ScanCells {C[0..1];} // Only the SCAN_CELL_NAME portion of the map
1093:     }
1094:     InheritNameMap C {
1095:       Prefix "C2";
1096:       ScanCells {C[2..3];}
1097:     }
1098:     InheritNameMap D {
1099:       Prefix "D1";
1100:       ScanCells {C[5..7];}
1101:     }
1102:     ScanCells {C[4] "FOO";}
1103:   }
1104: }
1105:
1106: Environment FLAT {
1107:   // This is a flat representation of NameMap for scan cells
1108:   // This mapping corresponds to the above HIERARCHICAL mapping
1109:   NameMaps {
1110:     ScanCells {
1111:       C[0] "C1/CELL1";
1112:       C[1] "C1/CELL2";
1113:       C[2] "C2/CELL1";
1114:       C[3] "C2/CELL2";
1115:       C[4] "FOO";
1116:       C[5] "D1/C3/CELL1";
1117:       C[6] "D1/C3/CELL2";
1118:       C[7] "D1/CELL3";
1119:     }
1120:   }
1121: }

```

## 19. Pragma block

A Pragma is a block of code that is implementation dependent. A Pragma block is a means of embedding non-STIL code within a STIL file/stream. As with standard annotations, the syntax within the pragma is not defined or limited in any way, except by the opening and closing brace/asterisk convention. A Pragma block may be used anywhere in a file/stream that an annotation is allowed (See STIL.0, Clause 13). The use of a Pragma block is dependent on the application understanding the specific named Pragma; a Pragma with a name not be understood when the application is ignored.

The Pragma block supports application-specific constructs outside the definitions of this standard, for example, to embed external functionality such as a fragment of C-code or Perl-code as part of the information present for a STIL test, or to embed test-specific code understood by an application as part of STIL test data.

Be aware that the use of the Pragma block is extremely application-specific. These data are primarily intended to instruct specific applications on how to apply STIL constructs, for example, to identify test resource allocations for a specific test environment. Functionality necessary for the correct operation of a STIL test program should not appear in a Pragma block as the interpretation of that functionality is nonstandard and the application of Pragma constructs are inherently nonportable.

### 19.1 Pragma syntax

**Pragma** (NAME) {\* ... APPLICATION DEPENDANT SYNTAX ... \*} (1)

- (1) **Pragma:** Keyword identifying an implementation-dependent block of code. If the block is used at the top level of a STIL file/stream, then it has global scope. As with UserKeywords, a pragma block is locally scoped to a containing STIL block.

NAME: The NAME is an optional identifier and if present is used to identify the application that is expected to process the contents of this block. The name is not required to be unique (i.e., all Pragma blocks for a given application should have the same NAME).

{\* ... APPLICATION DEPENDANT SYNTAX ... \*}: The content of the block is enclosed within brace/asterisk delimiters. The only restriction on this block of data is that it not contain the asterisk/brace character pair, which shall signal the end of the pragma data to an STIL parser.

## 20. PatternFailReport

The PatternFailReport block is used to report fail information resulting from the test of a device. As such, it is typically not part of a STIL test program file/stream, but it contains references to the associated STIL test program file/stream to establish the context. Each PatternFailReport block shall contain the set of fails from the test of one device on the tester.

See Clause 16 for the definition of the X (cross-reference) pattern statement, which is the basis for referencing fail data back to the pattern data. For example code, see Annex I.

## 20.1 PatternFailReport syntax

```

PatternFailReport (REPORT_NAME) { (1)
  ( DeviceID "IDENTIFYING INFORMATION"; ) (2)
  ( TestConditions { } ) (3)
  ( Pattern PAT_NAME; ) (4)
  ( Pattern PAT_NAME {
    ( LogStart (X_REF_ID(X_REF_ID)*) (CYCLE_COUNT); )
    ( LogStop (X_REF_ID(X_REF_ID)*) (CYCLE_COUNT); )
  } )
  ( PatternBurst PAT_BURST_NAME; ) (5)
  ( PatternBurst PAT_BURST_NAME {
    ( LogStart (X_REF_ID(X_REF_ID)*) (CYCLE_COUNT); )
    ( LogStop (X_REF_ID(X_REF_ID)*) (CYCLE_COUNT); )
  } )
  ( PatternExec PAT_EXEC_NAME; ) (6)
  FailData { (7)
    ( X_REF_ID(X_REF_ID)*) (8)
      SIG_NAME
      <
        ( CYCLE_OFFSET(.COMPARE_OFFSET) ) * ;
        | "observed_data" ;
        | ( CYCLE_OFFSET(.COMPARE_OFFSET) "observed_data" ) * ;
      >
    ) * // end fail data record
  } // end FailData
} // end PatternFailReport

```

- (1) **PatternFailReport**: The block contains fail data as produced on a tester and is associated with a STIL test program stream. The REPORT\_NAME is an optional name that can be assigned to the block. This block follows the standard rules for STIL blocks; there can be one un-named block and any number of named blocks that all shall have unique names.
- (2) **DeviceID**: This statement contains a string for the purpose of identifying the device-under-test that produced the reported results.
- (3) **TestConditions** { } : This block contains statements that identify the conditions under which the test was made. The test conditions would typically identify things such as environmental conditions like temperature; DC set p conditions like voltage and current; timing setup conditions; and ATE type and system identification. These statements shall either be Ann statements, or else user keyword statements.
- (4) **Pattern**: This statement specifies the name of the pattern in the associated STIL test program file/stream that is detecting the device failures. The block form of this statement allows for the specification of start and stop points of the logging information in this fail report. The X\_REF\_ID parameter refers to an X statement in the pattern. The CYCLE\_COUNT parameter represents the number of vector cycles (i.e., periods) that transpire to the commencement or termination of the logging function. When only the CYCLE\_COUNT is present, the cycle offset is from the beginning of the pattern. If both X\_REF\_ID and cycle count are present, the cycle offset is from the specified X\_REF\_ID.
- (5) **PatternBurst**: This statement specifies the name of the pattern burst in the associated STIL test program file/stream that is detecting the device failures. The block form of this statement allows for the specification of start and stop points. See the definition of these parameters in the Pattern statement.
- (6) **PatternExec**: This statement specifies the name of the pattern exec in the associated STIL test program stream that is detecting the device failures.
- (7) **FailData**: It begins the block containing the device failure data.

- (8) **fail data record:** Each device failure record contains the fail data information as described below.
- **X\_REF\_ID:** It is the required first token of each record, and it identifies the reference position within the pattern. It shall be a user-defined name according to the rules of STIL.0, Subclause 6.8, with one exception: the value 0. Reference shall always be to an X statement encountered in the pattern execution sequence within the base pattern. If there are X references within called procedures and/or macros, then the X\_REF\_ID is appended with a dot separator to the X\_REF\_ID in the base pattern. For patterns with no X statements, the value 0 (i.e., the integer zero, unquoted) can be used as an X\_REF\_ID to specify the offset from cycle 0 of the pattern or burst. All X\_REF\_ID names shall be unique within the pattern, burst, procedure, or macro context in which they are used.
  - **SIG\_NAME:** It is the name of the signal that detected the failure. It may be either a name in a Signals block or a renamed single signal in a SignalGroups block. The signal name resolution follows the standard rules for domain name resolution. The signal names always refers to the names as used in the referenced pattern; however, a Signals/SignalGroups block may be included in the fail data file/stream for the purpose of defining the hex formatting of the fail data records.
  - **CYCLE\_OFFSET:** This token indicates the cycle offset from the last X statement encountered in the pattern execution sequence. It is an optional field, and if not specified, a cycle offset of 0 is assumed. The cycle number for the X\_REF\_ID tagged vector shall be 0. If the last X statement encountered in the pattern execution sequence is within a Loop or Shift block, then the cycle offset is from the first occurrence of the X\_REF\_ID in the loop or shift.
  - **COMPARE\_OFFSET:** It is an optional token used when there are multiple compares within a cycle. If it is not specified, the first compare in the cycle is referenced. The first compare event in a cycle shall be compare event 0.
  - *observed\_data:* This field is optional and specifies the observed state(s) of the failing signal. The observed data are a string of WFC (or event) characters enclosed in double quotes. In full data capture mode, this is a string of observed states, with a state specified for each cycle (either H/L/T for failing cycles, or X for cycles for which no compare is done). In the case in which there are multiple compares in a cycle, then an observe state for each compare is specified. Compaction of this observed data is supported by expressing the data using the \h, \r, and \e constructs as defined in STIL.0, Subclause 21.1.
  - If there is no Signals or SignalGroups blocks in the fail report file/stream, then by default, the observed\_data is an event list comprising the compare event characters L, H, X, and T (low, high, don't-care, and tri-state) as defined in STIL.0, Subclause 18.2, Table 10.
  - If there is a Signals or SignalGroups block in the fail report that defines the observed signal name, then formatting of the observed data is possible. The “Base Hex;” statement can be used to specify hex formatting, which thus allows more compact hex logging with four cycles represented by each hex character. Also possible is the mapping to specific WFCs (see Annex I).

## 20.2 PatternFailReport example

The following examples are of syntax constructs. See Annex N for a complete example of pattern fail report statements:

```

1122: STIL 1.0 { Design 2005; }
1123: Header {
1124:   Source "STD 1450.1-2005";
1125:   Ann { * sub-clause 20.2 * }
1126: }
1127:
1128: Signals {

```

```
1129: PO1 Out;
1130: PO2 Out;
1131: SO1 Out; SO2 Out; SO3 Out; SO4 Out;
1132: SO5 Out { Base Hex LHT; }
1133: }
1134:
1135: PatternFailReport {
1136:   Pattern PAT;
1137:   PatternBurst BRST;
1138:   PatternExec EXE;
1139:   FailData {
1140:     X13 PO1; // example with pattern-unit identifier =X13; signal=PO1
1141:     X42 SO1 19; // example with cycle offset (i.e. scan cell in a shift)
1142:     X45.LU SO1 16; // example with reference inside a macro/proc (i.e the LoadUnload)
1143:     X46.M1.LU S01 16; // example of two levels of macro/proc
1144:     X59 PO2 23"H"; // example of fail state
1145:     X64 SO2 6 16 24 25 338; // example of multiple fails (5 in this case)
1146:     X72 SO3 13"H" 19"L" 45"L" 63"L"; // example of multiple fails with fail states
1147:     X88 S04 0 "HHHLLLLLHHH" 56 "HHHL"; // example of data capture format
1148:     0 PO1 19023"H"; // example referencing cycle 19023 from the beginning of the pattern
1149:     "ABIST-30" SO5 22 "52 \r20 00 A \e L"; // @ cycle 22: HHLT ...80 L's... TTL
1150:   } // end FailData
1151: } // end PatternFailReport
```

## Annex A

(informative)

### Glossary

The following terms and definitions are used within the context of this standard.

**A.1 boolean\_expr:** (1) A metatype used to present language constructs. (2) A reference to an expression that evaluates to a boolean: True or False (see 5.6).

**A.2 False:** The value 0 (zero) or a negative value returned from the evaluation of a boolean expression.

**A.3 integer\_expr:** (1) A metatype that presents language constructs. (2) A reference to an expression that evaluates to an integer value (see 5.7).

**A.4 integer\_list:** A list of space separated integers, or pairs of integers separated by the “..” operator (see 5.13).

**A.5 logic\_expr:** (1) A metatype that presents language constructs. (2) A reference to a logic expression in the context of a design model (see 5.8).

**A.6 real\_expr:** (1) A metatype that presents language constructs. (2) A reference to a real number that is expressed in either scientific notation (23.5, 19E-9) or in engineering notation (23.5 mV, 19 ns) (see 5.9).

**A.7 sigvar\_expr:** (1) A metatype that presents language constructs. (2) A reference to an expression containing WFC lists (see 5.11).

**A.8 time\_expr:** (1) A metatype that presents language constructs. (2) A reference to an expression that defines a time value (see 5.10).

**A.9 True:** A nonzero positive value returned from the evaluation of a boolean expression.

**A.10 WFC:** A single alphanumeric character representing a reference to a Waveform defined in the current WFT.

**A.11 WFC list:** A list or series of alphanumeric characters, each character representing a reference to a Waveform defined in the current WFT.

**A.12 WFT:** A STIL block statement that defines waveforms associated with each WFC for each signal.

## Annex B

(informative)

### Signal mapping using SignalVariables

#### B.1 Example of parameter passing to Macros (or Procedures)

STIL defines a mechanism for passing multiple bit data from a vector to a WFT (see STIL.0, Subclause 21.2). Using pattern expressions, the same parameter passing mechanism also can be applied to the parameters to a Macro or a Procedure. An example of this is as follows:

```

1152: STIL 1.0 { Design 2005; }
1153: Header {
1154:   Source "STD 1450.1-2005";
1155:   Ann { * sub-clause B.1 * }
1156: }
1157:
1158: Signals {
1159:   SIG[1..5] In;
1160: }
1161: Variables {
1162:   SignalVariable P1[4..0];
1163:   SignalVariable P2[4..0];
1164:   SignalVariable P3[4..0];
1165: }
1166: SignalGroups {
1167:   GRP = 'SIG[1..5]';
1168: }
1169:
1170: MacroDefs {
1171:   M1 {
1172:     C { P1 = #; P2 = #; P3 = #; }
1173:     V { GRP = \W P1 ; }
1174:     V { GRP = \W P2 ; }
1175:     V { GRP = \W P3 ; }
1176:   }
1177: }
1178:
1179: Pattern PAT1 {
1180:   Macro M1 { P1=11111; P2=00000; P3=11111; }
1181:   Macro M1 { P1=00000; P2=11111; P3=00000; }
1182: }

```

## B.2 Parameter passing with signal mapping

Vector data are passed from vectors to Macros or Procedures by means of signal group-oriented strings of waveform characters. A new capability in MacroDefs or Procedures allows the waveform characters from the parameters to be mapped onto other signal groups as required by the core test design interface. An example of a vector data expression is as follows:

```
1183: STIL 1.0 { Design 2005; }
1184: Header {
1185:   Source "STD 1450.1-2005";
1186:   Ann { * sub-clause B.2 * }
1187: }
1188:
1189: Signals {
1190:   X[0..27] InOut;
1191: }
1192:
1193: SignalGroups {
1194:   XBUS = 'X[0..27];
1195: }
1196: Variables {
1197:   SignalVariable P1[0..13];
1198:   SignalVariable P2[0..13];
1199: }
1200: MacroDefs {
1201:   M2 {
1202:     C { P1 = #; P2 = #; }
1203:     V { XBUS = \W P1[0..6] 1010HLHLXXXXXX \W P1[7..13]; }
1204:     V { XBUS = \W P2[0..6] 0000XXXXHLHLXX \W P2[7..13]; }
1205:     V { XBUS = \W P1[0..13] \W P2[0..13]; }
1206:   }
1207: }
1208:
1209: Pattern PAT2 {
1210:   Macro M2 { P1[0..13]=1111111XXXXXXX; P2[0..13]=0000000LLLLLLL; }
1211:   Macro M2 { P1[0..13]=1111000XXXXXXX; P2[0..13]=0000111HHHHLLL; }
1212: }
```

## B.3 A more complete example of parameters and signal variable expressions

- a) Definition of SignalVariables in a SignalGroups block
- b) Use of parameters in a Macro (or Procedure call)
- c) Use of # to update the parameter values
- d) Application of parameters in a Macro (or Procedure)
- e) Use of logic expressions (logic\_expr) with signal variables in a Macro (or Procedure)
- f) Process of reusing signal groups (unmapped) as signal variables (mapped) (Figure B.1)

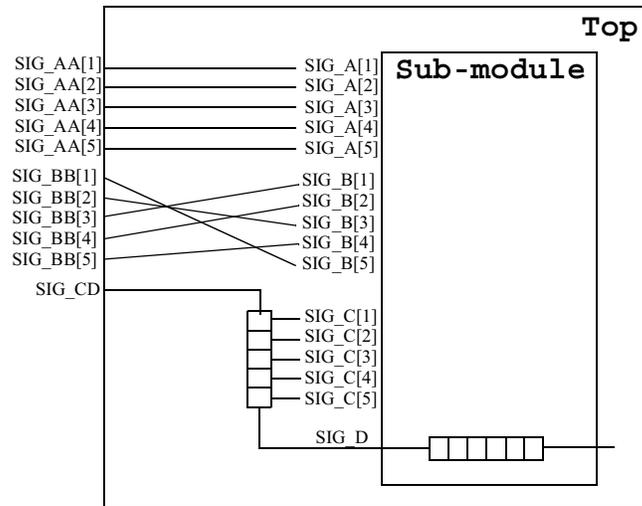


Figure B.1—Mapping of signals of a submodule

```

===== unmapped =====
1213: STIL 1.0 { Design 2005; }
1214: Header {
1215:   Source "STD 1450.1-2005";
1216:   Ann {* sub-clause B.3 - un-mapped *}
1217: }
1218: Signals {
1219:   SIG_A[1..5] In;
1220:   SIG_B[1..5] In;
1221:   SIG_C[1..5] In;
1222:   SIG_D In { ScanIn 6; }
1223: }
1224:
1225: SignalGroups {
1226:   GRP_C = 'SIG_C[1..5]';
1227: }
1228:
1229: MacroDefs {
1230:   mac_a {
1231:     V { SIG_A[1..5] = #; }
1232:     V { SIG_B[1..5] = #; }
1233:     V { GRP_C = #; }
1234:     Shift {
1235:       V { SIG_D = #; }
1236:     }
1237:   }
1238: }
1239: Include pattern.stil;
1240:
1241: ===== mapped =====
1242: STIL 1.0 { Design 2005; }

```

```

1243: Header {
1244:   Source "STD 1450.1-2005";
1245:   Ann {* sub-clause B.3 - mapped*}
1246: }
1247: Signals {
1248:   SIG_AA[1..5] In;
1249:   SIG_BB[1..5] In;
1250:   SIG_CD In { ScanIn 11; }
1251: }
1252:
1253: Variables {
1254:   SignalVariable SIG_A[1..5];
1255:   SignalVariable SIG_B[1..5];
1256:   SignalVariable GRP_C[1..5];
1257:   SignalVariable SIG_D[1..6];
1258: }
1259:
1260: MacroDefs {
1261:   mac_a {
1262:     C { SIG_A = #; SIG_B = #; GRP_C = #; SIG_D = #; }
1263:     V { SIG_AA[1..5] = \W SIG_A[1..5]; }
1264:     V { SIG_BB[1..5] = \W SIG_B[5 3 1 2 4]; }
1265:     // V { SIG_BB[3 4 2 5 1] = \W SIG_B[1..5]; } // alternate stmt
1266:     C { GRP_C[5 4 3 2 1] = \W GRP_C; }
1267:     Shift {
1268:       V { SIG_CD = \W SIG_D[1..6] \W GRP_C[1..5] ; }
1269:       //V { SIG_CD = \W SIG_D[1..6] 00000 ; } // alternate stmt
1270:       //V { SIG_CD = 00000 \W GRP_C[1..5] ; } // alternate stmt
1271:     }
1272:   }
1273: }
1274: Include pattern.stil;
1275:
===== pattern =====
1276: STIL 1.0 { Design 2005; }
1277: Header {
1278:   Source "STD 1450.1-2005";
1279:   Ann {* sub-clause B.3 - pattern include file *}
1280: }
1281: Pattern PAT_A {
1282:   Macro MAC_A {
1283:     SIG_A[1..5] = 10101;
1284:     SIG_B[1..5] = 11011;
1285:     GRP_C = 01010;
1286:     SIG_D = 001001;
1287:   }
1288: }

```

## Annex C

(informative)

### Using logic expression with signals

This example illustrates the following capabilities:

- a) Logic expression using & (and) operation
- b) Logic expression made up of signals
- c) Application of a logic\_expr to define a ScanEnable condition

```
1289: STIL 1.0 { Design 2005; }
1290: Header {
1291:   Source "STD 1450.1-2005";
1292:   Ann {* annex C *}
1293: }
1294:
1295: Signals {
1296:   SIG1 In;
1297:   SIG2 In;
1298: }
1299:
1300: ScanStructures STRUCT1 {
1301:   ScanChain CHAIN1 {
1302:     ScanLength 10;
1303:     ScanCells cell[1..10];
1304:     ScanEnable SIG1&SIG2;
1305:   }
1306: }
```

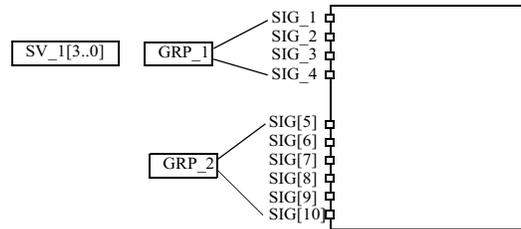
## Annex D

(informative)

### Using boolean expressions in patterns

This example illustrates the following capabilities:

- Use of parameters on a Macro (or Procedure)
- Use of boolean expressions with WFC data in conditional-If/Else statements in a pattern
- Use of parameters in boolean expression to create WFC data in a vector (Figure D.1)



**Figure D.1—Representation of signal groups for a design**

```

1307: STIL 1.0 { Design 2005; }
1308: Header {
1309:   Source "STD 1450.1-2005";
1310:   Ann { * annex D * }
1311: }
1312: Signals {
1313:   SIG_1 In; SIG_2 In; SIG_3 In; SIG_4 In;
1314:   SIG[5..10] In;
1315: }
1316: SignalGroups {
1317:   GRP_1 = 'SIG_1 + SIG_2 + SIG_3 + SIG_4';
1318:   GRP_2 = 'SIG[5..10]';
1319: }
1320: Variables {
1321:   SignalVariable SV_1[3..0];
1322: }
1323: MacroDefs {
1324:   MAC_1 {
1325:     C { SIG_1 = #; SIG_2 = #; }
1326:     If (\W SIG_1 == 1) { V { SIG_3 = A; } }
1327:     If ((\W SIG_1 == 1) && (\W SIG_2 == 0)) { V { SIG_3 = B; } }
1328:   }
1329:   MAC_2 {
1330:     C { GRP_1 = #; SV_1 = #; }
1331:     If (\W GRP_1 == 1100) { V { GRP_2 = \W SV_1 111111; } }
1332:     Else { V { GRP_2 = \W SV_1 000000; } }
1333:   }
1334: }
1335: Pattern PAT_1 {
1336:   Macro MAC_1 { SIG_1 = 1; SIG_2 = 0; }
1337:   Macro MAC_2 { GRP_1 = 1100; SV_1 = 0011; }
1338: }

```

## Annex E

(informative)

### Variables and expressions in algorithmic patterns

This example illustrates the following capabilities:

- a) Use of LoopData block
- b) Using integer logic expression to control looping of patterns
- c) Using integer logic expressions to create algorithmic data on signals and groups

```

1339: STIL 1.0 { Design 2005; }
1340: Header {
1341:   Source "STD 1450.1-2005";
1342:   Ann {* annex E *}
1343: }
1344: Variables {
1345:   Integer LOOP_CNT;
1346:   Integer H1 ;
1347:   Integer H2;
1348: }
1349: Signals {
1350:   ABUS[8..1] In;
1351:   BBUS[8..1] Out;
1352:   CLKA In; CLKB In; CLKC In;
1353: }
1354: SignalGroups {
1355:   GRPA = 'ABUS[8..1]' { Base Hex LH; }
1356:   GRPB = 'BBUS[8..1]' { Base Hex LH; }
1357:   CLOCKS = 'CLKA + CLKB + CLKC';
1358:   ALL_SIGS = 'GRPA + GRPB + CLOCKS';
1359: }
1360: MacroDefs {
1361:   MCLK_SIG {
1362:     LoopData {
1363:       V { CLKA = #; CLKB = #; CLKC = #; } // loop until all data consumed on #
1364:     }
1365:   }
1366:   MCLK_GRP {
1367:     LoopData {
1368:       V { CLOCKS = #; } // loop until all data consumed on #
1369:     }
1370:   }
1371:   MLOOP {
1372:     C { LOOP_CNT = #; }
1373:     While (LOOP_CNT <> 0) {
1374:       V { CLKA = P; LOOP_CNT := 'LOOP_CNT-1'; }
1375:     }
1376:   }
1377:   MLOOP2 {
1378:     C { LOOP_CNT := #; }
1379:     While (LOOP_CNT) {

```

```

1380:      C { LOOP_CNT := LOOP_CNT-1; }
1381:      V { CLKA = P; }
1382:    }
1383:  }
1384:  MDATA {
1385:    C { H1 := #; H2 := #; }
1386:    V { GRPA = 00001111; GRPB = 11110000; }
1387:    V { GRPA = \W H1; }
1388:    V { GRPB = \W H2; }
1389:  }
1390: } //end MacroDefs
1391:
1392: Pattern PAT1 {
1393:   C { ALL_SIGS = X; }
      // the following macro call generates 6 vectors
1394:   Macro MCLK_SIG { CLKA = PP0000; CLKB = 00PP00; CLKC = 0000PP; }
      // the following macro call generates 6 vectors
1395:   Macro MCLK_GRP { CLOCKS { P00; P00; 0P0; 0P0; 00P; 00P; } }
      // the following macro call generates 6 vectors
1396:   Macro MCLK_GRP { CLOCKS { P00P000P00P000P00P; } }
1397:   Macro MLOOP { LOOP_CNT := 100; }
1398:   Macro MLOOP2 { LOOP_CNT := 100; }
1399:   Macro MDATA { H1 := 0xFF; H2 := 0x0C; }
1400: }

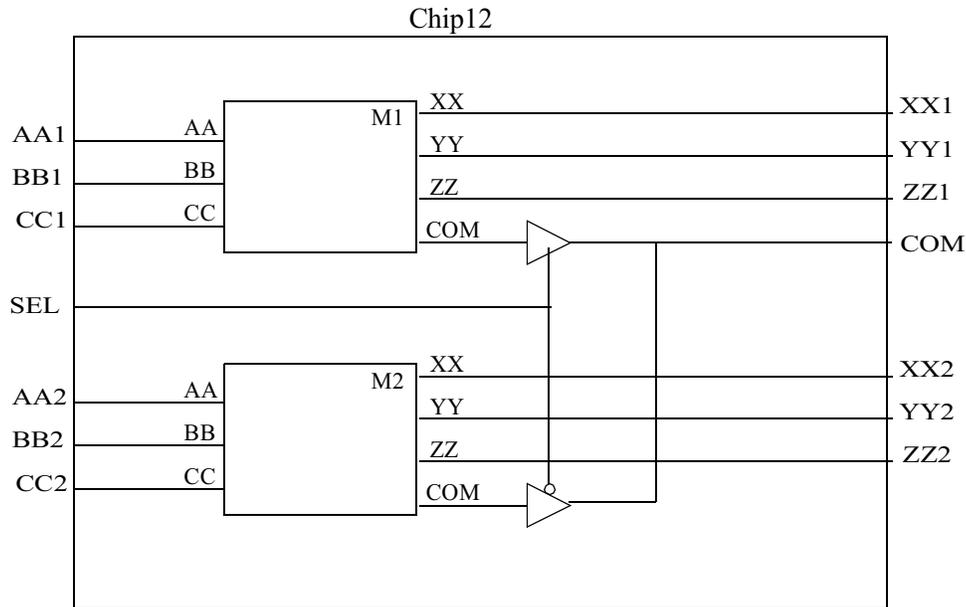
```

## Annex F

(informative)

### Using AllowInterleave

This example integrates two identical modules and integrates them at the chip level. A common signal is tied together at the chip level, and the AllowInterleave statement controls and monitors this signal from each of the two modules, in turn (Figure F.1).



**Figure F.1—Design containing two modules with a shared output**

The two modules (with instance names M1 and M2) are, in this case, identical modules. A single pattern (named PAT) defines the vectors required to test each instance of the module. The following STIL file defines the vectors to be run on the module, assuming the module is a stand-alone entity. The thing that is unusual about this pattern is that when the signal COM is to be compared, it is done in a macro (named COMMON). The reason is that it is anticipated that this module is to be integrated into a larger design and the COM signal is to be shared.

```

Signals {
  AA In; BB In; CC In;
  XX Out; YY Out; ZZ Out;
  COM Out; }
}
SignalGroups {
  ABC = 'AA+BB+CC';
  XYZ = 'XX+YY+ZZ';
}
MacroDefs {
  COMMON {
    W W1;
  }
}

```

```

        V { COM=#; }
        C { COM=X; }
    }
}
Pattern PAT { /* same pattern as defined below */ }

```

When these two modules are integrated into the larger design (CHIP12), several changes need to be made. But the pattern is unmodified. The changes to make note of are as follows:

- a) The Signals block is changed to define the signal names as they appear at the chip level. The three input signals (AA, BB, CC) have unique chip level names and the three output signals (XX, YY, ZZ). The signal COM can retain its same name at the chip level.
- b) Two SignalGroups blocks define the mapping of the chip signals to their respective module names. The name of the SignalGroups block is called out on the PatternBurst and thereby allows the same pattern to be used for both modules.
- c) The PatternBurst block has several things to note. The first is that two versions of PAT are executed together using the ParallelPatList statement and that the attribute LockStep is applied. It means that every vector in each pattern is to have a similar vector in the other pattern. Each instance of PAT has a specified SignalGroups domain defining the signals it is controlling and a MacroDefs domain defining the macro operation.
- d) Two MacroDefs blocks define the operation to be performed on the respective modules. It is within the macro named COMMON where the handling of the shared signal is managed. Note that the first statement in the macro is AllowInterleave and specifies the signal names SEL and COM. This statement allows the sharing of these two signal between the two parallel invocations of the macro COMMON. Note also that both definitions of COMMON have exactly the same number of vectors: two. In the first vector, the macro for M1 defines the SEL signal to be a 1 and compares the COM signal for the state as passed in from the pattern. In the first vector for module M2, the vector is empty, which thus allows M1 to control the activity on SEL and COM. In the second vector of the two macros, the opposite occurs. The second vector for module M2 is defining the activity on SEL and COM, whereas the second vector for M1 is empty.

```

1401: STIL 1.0 { Design 2005; }
1402:   Header {
1403:     Source "STD 1450.1-2005";
1404:     Ann { * annex F * }
1405:   }
1406: Signals {
1407:   AA1 In; BB1 In; CC1 In; AA2 In; BB2 In; CC2 In;
1408:   XX1 Out; YY1 Out; ZZ1 Out; XX2 Out; YY2 Out; ZZ2 Out;
1409:   COM Out;
1410: }
1411: SignalGroups SG_M1 {
1412:   ABC = 'AA1+BB1+CC1';
1413:   XYZ = 'XX1+YY1+ZZ1';
1414: }
1415: SignalGroups SG_M2 {
1416:   ABC = 'AA2+BB2+CC2';
1417:   XYZ = 'XX2+YY2+ZZ2';
1418: }
1419: Timing {
1420:   WaveformTable W1 { /* details omitted */ }
1421:   WaveformTable WCOM { /* details omitted */ }
1422: }

```

```

1423: MacroDefs MAC_M1 {
1424:   COMMON {
1425:     AllowInterleave 'SEL+COM';
1426:     W WCOM;
1427:     V { SEL=1; COM=#; }           // compare COM signal from M1
1428:     V {}                          // SEL, COM undefined
1429:     C { COM=X; }                 // return from macro with X state
1430:     W W1;
1431:   }
1432: }
1433: MacroDefs MAC_M2 {
1434:   COMMON {
1435:     AllowInterleave 'SEL+COM';
1436:     W WCOM;
1437:     V {}                          // SEL, COM undefined
1438:     V { SEL=0; COM=#; }           // compare COM signal from M2
1439:     C { COM=X; }                 // return from macro with X state
1440:     W W1;
1441:   }
1442: }
1443: PatternBurst BRST {
1444:   ParallelPatList LockStep {
1445:     PAT {SignalGroups SG_M1; MacroDefs MAC_M1;}
1446:     PAT {SignalGroups SG_M2; MacroDefs MAC_M2;}
1447:   }
1448: }
1449: Pattern PAT {
1450:   W W1;
1451:   C { ABC=000; XYZ=XXX; SEL=0; COM=X; }
1452:   V { ABC=001; XYZ=LLH; }
1453:   Macro COMMON { COM=H;}
1454:   V { ABC=011; XYZ=LHH; }
1455:   Macro COMMON { COM=L;}
1456: }

```

## Annex G

(informative)

### Vector data mapping using \m

Here is an example of the use of the signal mapping operation. See 15.2 for the definition of syntax and semantics.

#### G.1 Use case #1

Inversion of a signal is a common reason for using the map function. The characteristics of the inverted waveform can be completely defined in the new waveform definition that is to replace the one originally called for. The following list is of the typical transformations that would cause waveform inversion:

```
D -> U   ForceDown -> ForceUp
U -> D   ForceUp -> ForceDown
L -> H   CompareLow -> CompareHigh
H -> L   CompareHigh -> CompareLow
l -> h   CompareLowWindow -> CompareHighWindow
h -> l   CompareHighWindow -> CompareLowWindow
Z -> N   ForceOff -> ForceUnknown
T -> X   CompareOff -> CompareUnknown
t -> X   CompareOffWindow -> CompareUnknown
N -> N   ForceUnknown -> ForceUnknown
X -> X   CompareUnknown -> CompareUnknown
```

The following example demonstrates this usage:

```
1457: STIL 1.0 { Design 2005; }
1458:   Header {
1459:     Source "STD 1450.1-2005";
1460:     Ann { * sub-clause G.1 * }
1461:   }
1462:
1463: Signals {
1464:   A[0..15] In;
1465:   B[0..15] Out;
1466:   XSIGS[0..15] InOut;
1467: }
1468:
1469: SignalGroups { X = 'XSIGS[0..15]'; }
1470:
1471: Pattern PAT {
1472:   V { A = 1111000011110000; B = 0000111100001111; }
1473:   V { X = \WA[0..7] \m\WB[8..15]; }
1474:   V { X = \WA[0..7] XXXX \m\WA[8..11]; }
1475:   V { X = 11110000 \m1100 1100; }
1476: }
```

## G.2 Use case #2

In certain scan test styles, it is necessary to measure the output of the DUT's bidirectional signals in one cycle and then to drive the same logical values on the same signals from the tester in the next cycle, while turning the internal drivers off. A test pattern to accomplish this could have the following format:

- a) Load scan chains
- b) Force values on primary inputs (all clocks are off, bidirectionals are driven by DUT)
- c) Measure primary outputs and bidirectionals (all clocks are off)
- d) Force values on primary inputs [values are the same as in cycle b), except the internal bidirectional drivers are turned off by deactivating a bidirectional\_control input], force values on bidirectionals [same logical values as measured in cycle c)]
- e) Pulse capture clock
- f) Unload scan chains

Turning off the internal bidirectional drivers in cycle d) avoids possible contentions that can result in cycle e) because of capturing new data into the state elements. A bidirectional contention develops if the internal driver and the tester drive opposite logical values on the same bidirectional pin. The additional data to be applied on bidirectionals in cycle d) are redundant [can be computed from the data of cycle c)]. This test style needs to be supported without adding extra data to the STIL patterns and without changing the WFCs in the patterns. Also, ATPG rules checking can verify the correctness of the patterns [e.g., the internal bidirectionals are turned off in cycle d)] before actually generating test data.

Even if ATPG-generated patterns are contention-free on all bidirectionals, both pre- and post-capture [cycle e)], this protocol may be required for test flows already designed with this protocol or if timing-related glitches that may temporarily cause contention on the bidirectionals are to be avoided.

It is important that the bidirectional control input turns off ALL internal bidirectional drivers in cycle d). Otherwise, a contention-free pattern could be transformed into a pattern with contentions by the very protocol that attempts to avoid bidirectional contentions! For example, consider the following example, where BIDI1 and BIDI2 are bidirectionals, and BIDI\_CTRL is an input that, when 0, turns off the internal driver of BIDI1, but not of BIDI2.

ATPG-generated, contention-free pattern:

- a) Load scan chains
- b) Force values on primary inputs and bidirectionals (force BIDI\_CTRL=1; BIDI1 = Z; BIDI2 = Z;)
- c) Measure primary outputs and bidirectionals (measure BIDI1=L; BIDI2=H;)
- d) Pulse capture clock [it has the effect of switching the internal drivers such as now both the BIDI1 and the BIDI2 internal drivers are driving 0. There is no contention, because the tester continues to drive Z on both bidirectionals, as in cycle b)]
- e) Unload scan chains

This pattern can be changed, after it has been generated, to match the LNI protocol:

- a) Load scan chains
- b) Force values on primary inputs and bidirectionals (force BIDI\_CTRL=1; BIDI1 = Z; BIDI2 = Z;)
- c) Measure primary outputs and bidirectionals (measure BIDI1=L; BIDI2=H;)
- d) Force values on primary inputs (force BIDI\_CTRL=0; BIDI1=0; BIDI2=1;)
- e) Pulse capture clock [It has the effect of switching the internal drivers such as now both the BIDI1 and the BIDI2 internal drivers are driving 0. This causes a *contention* on BIDI2: its internal driver, not turned off, drives 0 while the tester drives 1, as in cycle d)!]
- f) Unload scan chains

#### Example

```
1477: STIL 1.0 { Design 2005; }
1478: Header {
1479:   Source "STD 1450.1-2005";
1480:   Ann {* sub-clause G.2 *}
1481: }
1482: Signals { A In; CK In; BIDI_ENABLE In; B Out; Q1 InOut; Q2 InOut; }
1483:
1484: SignalGroups {
1485:   _IO = 'Q1+Q2' {
1486:     WFCMap {
1487:       L->0; H->1; T->Z; X->N;
1488:     }
1489:   }
1490: }
1491:
1492: PatternBurst "_BURST_" {
1493:   PatList {"_PATTERN_";}
1494: }
1495:
1496: Procedures PROCDOMAIN {
1497:   "CAPTURE_SYSClk" {
1498:     W MYWFT; // where all WaveformCharacters are defined
1499:     "cycle 2": V { A=#; CK=0; BIDI_ENABLE=1; B=X; _IO=ZZ ; }
1500:     "cycle 3": V { B=#; _IO=%%; }
1501:     "cycle 4": V { BIDI_ENABLE=0; B=X; _IO=\m ##; }
1502:     "cycle 5": V { CK=P; }
1503:   }
1504: }
1505:
1506: Pattern "_PATTERN_" {
1507:   W myWFT;
1508:   "cycle 1": Call "LOAD_UNLOAD" { }
1509:   Call "CAPTURE_SYSClk" { A=0; B=H; _IO=HL; }
1510:   "cycle 6": Call "LOAD_UNLOAD" { }
1511: }
```

In this example, the vectors are labeled to correspond to the cycles above. Cycle c) uses the arguments passed in for \_IO first (HL), and then cycle d) uses them again, this time mapped to (10), which remain in effect for cycle e) as well.

## Annex H

(informative)

### Vector data joining using \j

The join operation allows multiple WFCs against the same signal in one vector. This annex is an application of this capability. See 15.2 for details of the syntax and semantics.

#### H.1 Example

The following is an example usage of the join function:

```

1512: STIL 1.0 { Design 2005; }
1513: Header {
1514:   Source "STD 1450.1-2005";
1515:   Ann { * sub-clause H.1 * }
1516: }
1517: Signals {
1518:   B InOut {
1519:     WFCMap {
1520:       0x -> k;
1521:       // The 2 source WFCs are not order-sensitive.
1522:       // The above could also be written as: { x0 -> k; }
1523:     }
1524:   }
1525: }
1526: PatternBurst "_BURST_" {
1527:   PatList {P;}
1528: }
1529: Pattern P {
1530:   V { B = \j 0; B = \j x; } // Using mapping above, this is: V{B = k;}
1531:   V { B = \j 1; } // This is: V{B = 1;}
1532: }

```

Although any signal or signal group may be mapped, it is more common to use the \j mapping to control WFC assignment to bidirectional (InOut) signals or signal groups. It enables structuring of STIL patterns using acter assignment to bidirectional (InOut) signals or signal groups. It enables structuring of STIL patterns using Procedures, as shown in Table H.1.

For instance, take the case in which two SignalGroups have a common element in them (signal 'b'):

```

_pi = '...+b';
_po = '...+b';

```

A procedure may join these two groups in a vector:

```

Procedure { cs { v { _pi= \j #; _po= \j #; }}}

```

Signal 'b' needs to be resolved based on the combinations of WFCs that may be seen by these two groups. It might have a WFCMap declaration (in the definition of "b"), like

```

WFCMap { 0x -> 0; 1x -> 1; }

```

**Table H.1—Example of “two data” conditions on an InOut Signal**

Force	Measure
0, 1, Z, N	X
Z	L, H, T
0	L
1	H
0	H, T
1	H, T

This mechanism provides for the explicit resolution of joined data without creating new combinations of waveforms on-the-fly.

## H.2 Usage example

Consider a design with one input, one output, and two bidirectionals, which is declared as follows:

```
Signals { I In; O Out; B1 InOut; B2 InOut; }
```

Signal groups are also defined as follows:

```
Signalgroups {
  _PI = 'I + B1 + B2';
  _PO = 'O + B1 + B2';
  _IO = 'B1 + B2';
}
```

Patterns are written out using capture procedures defined as follows:

```
Procedures {
  "capture" {
    V { _PI=### ; _PO=###; }
  }
}
```

Consider an ATPG test pattern that includes the following cycle:

```
force_all_pis: I=0; B1=Z; B2=1;
measure_all_pos: O=H; B1=H; B2=X;
```

In STIL, it would be written as follows:

```
Call capture { _PI=0Z1; _PO=HHX; }
```

Understanding how STIL is interpreted by the consumer of the patterns, the actual arguments are substituted for the formal arguments # in the body of the procedure, and the signalgroups `_pi` and `_po` are expanded to their signals, which results in

```
V { I=0; B1=Z; B2=1; O=H; B1=H; B2=X; }
```

However, multiple WFCs assigned to the same signal in a given vector create ambiguity. Thus, the above vector becomes

```
V { I=0; /* B1, B2 ambiguous */ O=H; }
```

The IEEE 1450.1 solution is very simple and general: Provide a mapping to explicitly explain what to do with the two WFC assignments. Thus, the IEEE 1450.1 procedure would be written as follows:

```
Procedures {
  CAPTURE {
    V { _PI= \j ### ; _PO= \j ###; }
  }
}
```

Notice the addition of the join modifier \j. The \j refers to the WFCMap mapping table that could be defined as follows:

```
SignalGroups {
  _PI= ' ... ' {
    WFCMap {
      0X -> 0; 1X -> 1; ZX -> Z; NX -> N; // bi-directional as input
      ZL -> L; ZH -> H; ZT -> T; // bi-directional as output
    }
  }
  _PO= ' ... ' {
    WFCMap {
      0X -> 0; 1X -> 1; ZX -> Z; NX -> N; // bi-directional as input
      ZL -> L; ZH -> H; ZT -> T; // bi-directional as output
    }
  }
}
```

This example provides an unambiguous interpretation of the above:

```
Call capture { _PI=0Z1; _PO=HHX; }
```

to the desired:

```
force_all_pis { I=0; B1=Z; B2=1; }
measure_all_pos ( O=H; B1=H; B2=X; )
```

## Annex I

(informative)

### Block data collection

This annex describes a usage scenario that combines several capabilities defined in this standard to provide complete support for returning tester information for evaluation by EDA tools. Although this example is oriented around returning information for diagnostic applications, this functionality is not constrained to this application. This scenario can also be applied to the general notion of learn-mode tester operation, where the test provides the stimulus and the response is learned by reporting the behavior from testing known-good devices. Integration of the return values in this mode of operation into a new test program is the responsibility of the tool that interprets the returned data.

To support the return of test values from a STIL test program, several constructs need to be present in the program. This example assumes a scenario in which the desired return information is a subset of both the set of signals in the design (that is, only one Signal needs to be evaluated) and a subset of the test vector sequence (that is, there is a specific region of the test where this information is needed).

#### I.1 Step 1—Identifying signals

The set of signals that return response data are identified by defining a waveform for those signals that contains an S or CompareSubstitute event for the region where the value needs to be checked. Typically, the timing of the S event will coincide with the timing for CompareLow and CompareHigh checks also defined (but may not be used) for that Signal. The following example is the WFT definition containing the CompareSubstitute:

```
Timing ALT {
  WaveformTable A {
    Period '20ns';
    Waveforms {
      A1 { 01XC { '14ns' L/H/X/S; } }
    }
  }
}
```

If the timing of the CompareLow check does not match the timing of the CompareHigh check, a decision is made on how to define the timing of the waveform containing the S event. Typically, the S event would be defined in a waveform that defined the overlapping region of both the low strobe and the high strobe, to return the correct value if either of these states was detected during the test execution.

#### I.2 Step 2—Identifying response waveforms

Several waveforms may have the same or similar characteristics to the waveform containing the S event. To specify the WFC that represents the return value from the S event, the WFCMap construct is used, which specifies an inverse mapping relationship. It is specified for this context on Signal A1 as follows:

```
Signals {
  A1 Out { WFCMap { C -> 01; } }
}
```

Notice that the order of WFC references in the mapped output side (the right or TO\_WFC side) corresponds to the order in the timing-waveform definition: the '0' represents the compare-low state and the '1' the compare-high state.

If there are multiple waveform definitions, each with S events for this Signal, each different waveform (with a unique WFC reference) can specify a unique set of waveform response mappings. It allows for differentiation of the response data if desired or needed for evaluation/review purposes.

### I.3 Step 3—Pattern reference

The final step is to identify the desired pattern where response data are captured. It is indicated by the application of the WFC that contains an S event, in the Pattern data. One example is as follows:

```
Pattern DIAG {
    ...
    X BLOCKREAD;
    Loop 200 { V { CLK=P; A1=C; } }
    ...
}
```

It enables the response-return for all cycles inside this single vector loop construct (in this example, for 200 cycles).

### I.4 Step 4—Tester response return

The return format for this information is defined in the PatternFailReport block (Clause 20). An example of this data is as follows:

```
1533: STIL 1.0 { Design 2005; }
1534: Header {
1535:     Source "STD 1450.1-2005";
1536:     Ann { * sub-clause I.4 * }
1537: }
1538: Signals { A1 Out { WFCMap { C -> 01; } } }
1539: Timing {
1540:     WaveformTable A {
1541:         Period '20ns';
1542:         Waveforms {
1543:             A1 { 01XC { '14ns' L/H/X/S; } }
1544:         }
1545:     }
1546: }
1547: PatternFailReport {
1548:     PatternExec EXEC;
1549:     PatternBurst BURST;
1550:     Pattern DIAG;
1551:     FailData {
1552:         BLOCKREAD A1 0
1553:         "0000000000 1111111111 0000000000 1111111111 0000000000
1554:         1111111111 0000000000 1111111111 0000000000 1111111111
1555:         0000000000 1111111111 0000000000 1111111111 0000000000
1556:         1111111111 0000000000 1111111111 0000000000 1111111111";
1557:     } //end FailData
1558: }
```

## Annex J

(informative)

### Using Fixed and Equivalent statements

This annex shows an example application using the Fixed and Equivalent statements as signal constraints. See 16.2 for the definition of syntax and semantics.

#### J.1 Example: Signal constraints

The following example is of Fixed and Equivalent statements used in Pattern cpat. For the first two vectors, signals a and b must be '0' and signals c and d must be either '0' and '1' or '1' and '0' or have both WFCs other than '0' and '1':

```

1559: STIL 1.0 { Design 2005; }
1560: Header {
1561:   Source "STD 1450.1-2005";
1562:   Ann { * sub-clause J.1 * }
1563: }
1564: Signals {
1565:   A In; B In; C In; Q Out;
1566:   D In { WFCMap { 0->1; 1->0; } }
1567: }
1568:
1569: SignalGroups { CINPUTS = 'A+B'; }
1570:
1571: Timing {
1572:   WaveformTable MYWFT {
1573:     Period '10ns';
1574:     Waveforms {
1575:       'CINPUTS+C+D' { 01ZN { '0ns' D/U/Z/N; } }
1576:       Q { LHTX { '0ns' X; '4ns' L/H/T/X; } }
1577:     }
1578:   }
1579: }
1580:
1581: PatternBurst BURST {
1582:   PatList { CPAT; }
1583: }
1584:
1585: Pattern CPAT {
1586:   W MYWFT;
1587:   Fixed { CINPUTS=00; }
1588:   Equivalent C \m D;
1589:   V { C=0; Q=H; } // equivalent to V { A=0; B=0; C=0; D=1; Q=H; }
1590:   V { C=1; Q=L; } // equivalent to V { A=0; B=0; C=1; D=0; Q=L; }
1591:   V { D=1; } // equivalent to V { A=0; B=0; C=1; D=1; Q=L; }
1592:   Fixed { C=0; } // implies also Fixed { D=1; }
1593:   V {} // equivalent to V { A=0; B=0; C=0; D=1; Q=L; }
1594: }

```

## J.2 Example: Constraints in a scan design

The following example is of a TEST\_MODE signal, which is typical in a scan design that needs to be a constraint when applying values on primary inputs, but not during load/unload of the scan chains. The Fixed statement is active in procedure CAPTURE\_SYSCLK, but not in procedure LOAD\_UNLOAD. Also, the design has differential scan inputs SI1 and SI1B; thus the Equivalent statement is active in procedure LOAD\_UNLOAD, but not in CAPTURE\_SYSCLK:

```

1595: STIL 1.0 { Design 2005; }
1596: Header {
1597:   Source "STD 1450.1-2005";
1598:   Ann {* sub-clause J.2 *}
1599: }
1600: Signals {
1601:   TEST_MODE In;
1602:   SI1 In;
1603:   SI1B In { WFCMap { 0->1; 1->0; }}
1604: }
1605:
1606: PatternBurst BURST {
1607:   PatList { PATTERN; }
1608: }
1609:
1610: Procedures {
1611:   CAPTURE_SYSCLK {
1612:     W functionalWFT;
1613:     F { TEST_MODE=0; }
1614:     "force PI": V { }
1615:     "measure PO": V { }
1616:     "pulse clock" : V { }
1617:   }
1618:   LOAD_UNLOAD {
1619:     W scanWFT;
1620:     E SI1 \m SI1B;
1621:     Shift { V { }}
1622:   }
1623: }
1624:
1625: Pattern PATTERN {
1626:   W myWFT;
1627:   "cycle 1": Call LOAD_UNLOAD { }
1628:   Call CAPTURE_SYSCLK { ... }
1629:   "cycle 6": Call LOAD_UNLOAD { }
1630: }

```

## Annex K

(informative)

### Independent parallel patterns

The ParallelPatList as defined in the PatternBurst block is fine for the case in which the user wants a tightly coupled execution of patterns. A different, but related application is where there are two independent sets of signals, timing, and patterns on a device that are to be tested independently, with no regard for each other. Consider the following example, which defines a user keyword (ParallelTest<sup>6</sup>) for a test with two PatternExec reference statements (Figure K.1):

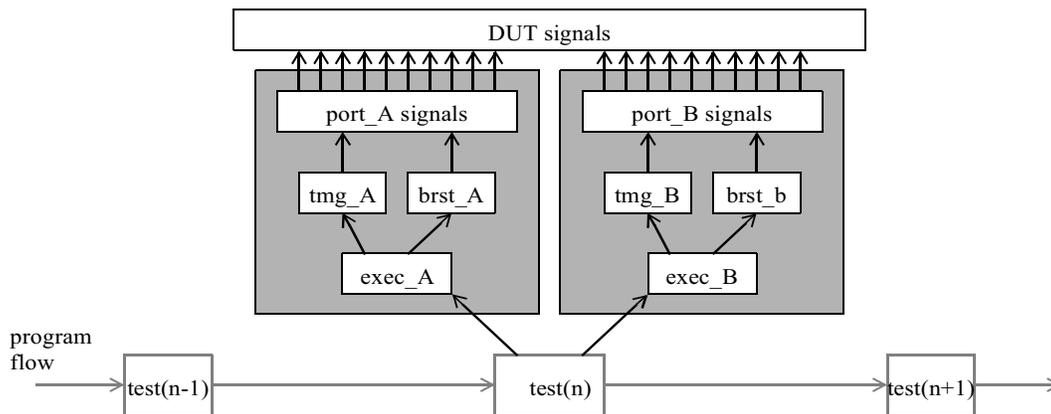


Figure K.1—Representation of a complete test program

```

1631: STIL 1.0 { Design 2005; }
1632: Header {
1633:     Source "STD 1450.1-2005";
1634:     Ann {* annex K *}
1635: }
1636: UserKeywords ParallelTest;
1637:
1638: Signals { DUT1 In; DUT2 In; DUT3 In; DUT4 In; }
1639: SignalGroups PORT_A { A1=DUT1; A2=DUT2; }
1640: SignalGroups PORT_B { B1=DUT3; B2=DUT4; }
1641:
1642: ParallelTest XYZ {
1643:     PatternExec EXEC_A;
1644:     PatternExec EXEC_B;
1645: }
1646: PatternExec EXEC_A {
1647:     PatternBurst BRST_A;
1648:     Timing TMG_A;
1649: }
1650: PatternExec EXEC_B {
1651:     PatternBurst BRST_B;

```

<sup>6</sup>The standardization of this technique is more appropriately done in the STIL extension that defines program flow and test methods (STIL.4, STIL.5).

```
1652:    Timing TMG_B;
1653: }
1654: PatternBurst BRST_A {
1655:     SignalGroups PORT_A;
1656:     PatList { PAT_A1; PAT_A2;}
1657: }
1658: PatternBurst BRST_B {
1659:     SignalGroups PORT_B;
1660:     PatList { PAT_B1; PAT_B2;}
1661: }
1662: Timing TMG_A { }
1663: Timing TMG_B { }
1664: Pattern PAT_A1 { }
1665: Pattern PAT_A2 { }
1666: Pattern PAT_B1 { }
1667: Pattern PAT_B2 { }
```

## Annex L

(informative)

### Applications using new ScanStructures syntax

The ScanStructures block is extended to include additional information required for efficient simulation of scan patterns, i.e., eliminating the need to serially simulate load/unload cycles. Refer to Clause 14 for the syntax definition. The ScanStructures block is optional in a STIL pattern file/stream because the data are tester-ready. However, the ScanStructures block may be used by a simulator to more efficiently simulate the load/unload of scan chains by loading and unloading scan cells in parallel.

Simulation of scan patterns outside the test-pattern generator is often performed to verify timing, design functionality, or the library used to generate the patterns. The speed of the simulator is limited by simulating the load/unload (Shift) operation of scan chains. Optimal simulation performance is achieved with no shifts, bypassing scan chain logic and asserting/verifying the scan data directly on the scan cells.

A scan cell may contain one or more state elements in various configurations. All state elements defined in one scan cell access a single scan bit value; one scan bit in the data stream is allocated to each scan cell construct defined. Stability and capture ability are identified for each state element in the scan cell, in addition to the inversion information with respect to both scan input and output by scan-oriented test pattern generators. For example, loading a single scan bit may require setting values in several state elements. Additionally, there may be extra cycles in the load or unload of only some test patterns, which thus shifts the data. A typical scan-pattern simulation process is summarized as follows:

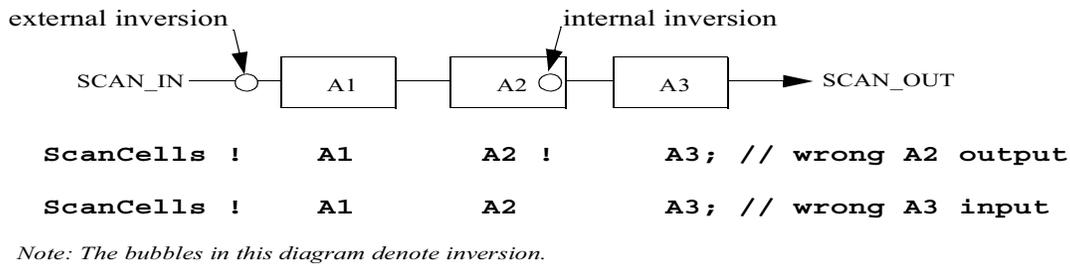
- a) Load: many cycles (Shift operation), not simulated in parallel load
- b) Skewed load: optional cycle, not simulated in parallel load, if possible
- c) Force values on primary inputs: simulated
- d) Measure values on primary outputs: simulated
- e) Pulse capture clock(s): optional cycle(s), simulated
- f) OBSERVE procedure: optional cycle, not simulated in parallel unload, if possible
- g) Unload: many cycles (Shift operation), not simulated in parallel load

The ScanStructures block is extended to include additional information required for efficient simulation of scan patterns, i.e., eliminating the need to simulate load/unload (Shift) cycles. Additional constructs are defined on the ScanCell statement inside the ScanChain block. In addition, the ScanChainGroups statement is added in the ScanStructures block to allow grouping of scan chains and to support referencing previous ScanChain definitions from other ScanStructure blocks.

#### L.1 Example: Inversion inside a scan cell

To use the tester-ready, serial data for parallel load/unload, the simulator must know the input and output inversions for every scan cell with respect to the scan-input and scan-output pins of the scan chain. In STIL, these inversions are represented by ‘!’ characters in the ScanCells list. However, this notation cannot unambiguously represent both input and output inversions for all scan cells if one or more cells have internal inversions, as in Figure L.1.

In Figure L.1, scan cell a2 has an internal inversion: The value scanned out of a2 is inverted with respect to the value scanned into a2.



**Figure L.1—Example of scan chain including a cell with internal inversion**

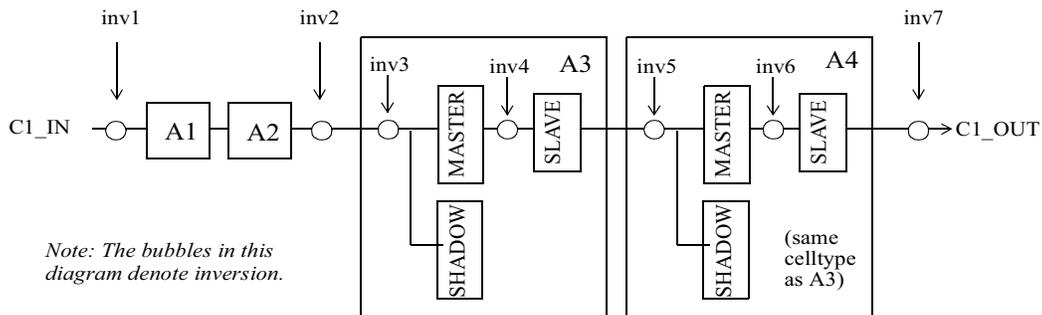
In the first ScanCells statement, the inversion between a2 and a3 implies that the output of cell a2 is inverted at the scan\_out pin, which is incorrect. The second ScanCells statement correctly represents the output of cell a2 (noninverted at scan\_out), but it incorrectly shows the input of a3 inverted with respect to the scan\_in pin, whereas in reality the two inversions cancel each other out and make the input of a3 noninverted with respect to scan\_in. There is no correct representation of this scan chain using this limited ScanCells notation.

Using the extensions to the ScanStructures block, Figure L.1 can be correctly represented as follows:

```
ScanCells { !; A1; A2 { CellOut ! } A3; }
```

## L.2 Example: Scan cells with multiple state elements

In Figure L.2, chain C1 is defined with four scan cells, cell A3 having a more complex definition. All inversions have been marked with a comment in the form /\*invn\*/ to facilitate explanations. Cell A3 resembles an LSSD cell, with the master shift clock being ACLK and the slave shift clock being BCLK. The other scan cells would typically have the same configuration as A3, but only A3 is detailed here for brevity.



**Figure L.2—Example of scan chain including a cell with multiple state elements and internal inversions**

This scan chain is defined as follows:

```
1668: STIL 1.0 { Design 2005; }
1669: Header {
1670:   Source "STD 1450.1-2005";
1671:   Ann { * sub-clause L.2 * }
1672: }
1673: // The simulation-only variable SCANMODE is declared as follows:
1674: Variables {
1675:   Integer SCANMODE {
```

```
1676:     InitialValue 2;
1677:   }
1678: }
1679: Signals { C1_IN In; C1_OUT Out; ACLK In; BCLK In;}
1680: ScanStructures G1 {
1681:   ScanChain C1 {
1682:     ScanIn C1_IN;
1683:     ScanOut C1_OUT;
1684:     ScanCellType MS {
1685:       If (SCANMODE > 0)
1686:         CellIn ! /*inv3*/ MASTER SHADOW ! /*inv4*/ SLAVE;
1687:       if (SCANMODE == 2)
1688:         CellOut SLAVE;
1689:       if (SCANMODE == 1)
1690:         CellOut MASTER ! /*inv4*/;
1691:     } // end ScanCellType
1692:     ScanCells { ! /*inv1*/; A1; A2; ! /*inv2*/; A3 MS; A4 MS; ! /*inv7*/; }
1693:   } // end ScanChain
1694: } // end ScanStructures
1695:
1696: // The variable SCANMODE controls simulation.
1697: // The load/unload procedures are setting this variable as follows:
1698: Procedures {
1699:   SKEWED_LOAD {
1700:     C { SCANMODE := 0; } // no parallel simulation
1701:     V { C1_IN = #; ACLK = P; BCLK = 0; } // pulse A-clock
1702:     C { SCANMODE := 2; } // reset
1703:   }
1704:   LOAD_UNLOAD {
1705:     // uses current value of SCANMODE
1706:     Shift {
1707:       V { C1_IN = #; C1_OUT = #; ACLK = P; BCLK = P; }
1708:     }
1709:     C { SCANMODE := 2; } // reset
1710:   }
1711:   MASTER_OBSERVE {
1712:     C { SCANMODE := 1; } // MASTER_OBSERVE mode
1713:     V { BCLK = P; ACLK = 0; } // pulse B-clock
1714:   }
1715: }
1716:
1717: /* The pattern block need not be concerned with the details of the scan chain
and does not explicitly change the variable SCANMODE.*/
1718: Pattern SCAN {
1719:   C { ACLK = 0; BCLK = 0; }
1720:   "pattern 1": Call LOAD_UNLOAD { C1_IN = 0001; } // SCANMODE==2
1721:   Call SKEWED_LOAD { C1_IN = 0; } // sets SCANMODE to 0, then 2
1722:   V { }
1723:   Call MASTER_OBSERVE; // sets SCANMODE to 1
1724:   "pattern 2": Call LOAD_UNLOAD { C1_OUT = 1110; } // SCANMODE==1
1725:   V { } // SCANMODE was set to 2 at the end of previous LOAD_UNLOAD
1726:   "pattern 3": Call LOAD_UNLOAD { C1_OUT = 1110; } // SCANMODE==2
1727: }
```

In the pattern shown, the first vector (labeled "pattern 1") calls the LOAD\_UNLOAD procedure. The simulator can execute a fully parallel load of { C1\_IN = 0001; } because SCANMODE is 2 (from its InitialValue in the PatternBurst). The first three bits applied on c1\_in are 0, and the last bit is 1. It results in the following values being loaded:

- A1 is the cell closest to the scan input c1\_in and is thus loaded with the last value (1) of the "0001" string, inverted as indicated by the "!" /\*inv1\*/; thus, A1 = 0.
- A2 is the next cell, loaded with the next-to-last value (0), also inverted /\*inv1\*/; thus, A2 = 1.
- A3 is loaded with 0 inverted twice /\*inv1\*/ /\*inv2\*/; within A3 the MASTER has yet another inversion /\*inv3\*/; thus, A3/MASTER = 1 and A3/SHADOW = 0.
- A4 is the cell closest to the scan output c1\_out and is thus loaded with the first value (0) inverted twice /\*inv1\*/ /\*inv2\*/; within A4, the MASTER has yet another inversion /\*inv5\*/; thus, A4/MASTER = 1 and A4/SHADOW = 0.

Next, the SKEWED\_LOAD procedure is called in the Pattern block. The { C1\_IN = 0; }; this procedure must be serially simulated because SCANMODE is set to 0 and will affect the values just loaded into the scan cells. At the end, procedure SKEWED\_LOAD sets SCANMODE back to 2.

The following vector is also simulated. Next, the MASTER\_OBSERVE procedure is called. This procedure has no parameters and only affects how the following call to LOAD\_UNLOAD (data { C1\_OUT = 1110; } and labeled "pattern 2") is to be interpreted, by setting SCANMODE to 1:

- A4 is the cell closest to the scan output C1\_OUT and is unloaded with the first value (1) of the "1110" string, inverted as indicated by the "!" /\*inv7\*/. Within A4, when SCANMODE = 1, the MASTER is unloaded with yet another inversion /\*inv6\*/; thus, A4/MASTER = 1;
- A3 is the next cell, unloaded with the second value (1), also inverted /\*inv7\*/. Within A3, when SCANMODE = 1, the MASTER is unloaded with yet another inversion /\*inv4\*/; thus, A3/MASTER = 1;
- A2 is the next cell, unloaded with (1) inverted twice /\*inv2\*/ /\*inv7\*/; thus, A2 = 1;
- A1 is the cell closest to the scan input C1\_IN and is unloaded with the last value (0) inverted twice /\*inv2\*/ /\*inv7\*/; thus, A1 = 0.

Next, LOAD\_UNLOAD is again called (label "pattern 3"). This time, SCANMODE is 2 (set at the end of the previous LOAD\_UNLOAD). The same unload data { C1\_OUT = 1110; } is now interpreted differently for cells A4 and A3:

- A4 is the cell closest to the scan output C1\_OUT and is unloaded with the first value (1) of the "1110" string, inverted as indicated by the "!" /\*inv7\*/. Within A4, when SCANMODE = 2, the SLAVE is unloaded; thus, A4/SLAVE = 0;
- A3 is the next cell, unloaded with the second value (1), also inverted /\*inv7\*/. Within A3, when SCANMODE = 2, the SLAVE is unloaded; thus, A3/SLAVE = 0;.

## Annex M

(informative)

### BreakPoints using MergedScan() function

STIL writers will primarily output STIL scan-based patterns using a merged format, being motivated by the test time-reduction savings. Typically, in a merged scan format, the unload operation of a scan pattern is merged in with the load operation of the next scan pattern. It results in effectively cutting in half the number of vectors applied.

As a result of outputting STIL patterns in merged format, no convenient location exists in the STIL patterns to place a STIL BreakPoint statement. For a BreakPoint statement to exist within STIL scan patterns that are in merged format, it would typically need to exist between the two unload/load operations that have been merged together. When processing large sets of patterns, a need exists to break those patterns up into smaller, more manageable, pattern segments. If the patterns being processed are merged scan patterns, then there is no standard way of breaking them up into independent patterns segments.

STIL BreakPoint statements are used to segment the patterns into logical chunks by identifying the points in the patterns where they may be broken up. Breaking up STIL patterns at the BreakPoint statement will ensure that the resulting pattern segments will each function independently of each other, and that these segments may be applied as a standalone entity to a tester or simulator.

By using the MergedScan() function, STIL writers can provide both merged and unmerged formats of scan patterns within a single pattern instance. Usage of either the merged or the unmerged formats is then left up to the STIL consumers to use on a pattern-by-pattern basis, depending on their processing requirements and resources. An example of this is shown as follows.

#### M.1 Example of merged and unmerged STIL scan patterns using MergedScan()

This example shows the usage of the MergedScan() function and how it may be applied in scan procedures or macros containing shift blocks.

First, the Signals and SignalGroups blocks defining the scanin and scanout signals and groups:

```
1728: STIL 1.0 { Design 2005; }
1729: Header {
1730:   Source "STD 1450.1-2005";
1731:   Ann { * sub-clause M.1 * }
1732: }
1733: Signals {
1734:   SI1 In{ScanIn 8;}
1735:   SI2 In{ScanIn 6;}
1736:   SO1 Out{ScanOut 8;}
1737:   SO2 Out{ScanOut 6;}
1738:   CLK In; CLKS[1..3] In;
1739:   PI[1..11] In;
1740:   PO[1..11] Out;
1741: }
1742:
1743: SignalGroups {
```

```

1744:    SI = 'SI1 + SI2';
1745:    SO = 'SO1 + SO2';
1746:    CLKS = 'CLKS [1..3]';
1747:    PI = 'PI [1..11]';
1748:    PO = 'PO [1..11]';
1749: }

```

Next, a Procedures block defines a single scan procedure to be used for all scan operations, both merged and unmerged. Note the If/Else logic based on the result of a call to the boolean function MergedScan(). It is the key area of the STIL patterns that allows for either a merged or an unmerged representation of scan patterns. The If/Else constructs will ensure a mutually exclusive relationship between these two formats such that the STIL consumer will always apply one and only one scan format:

```

1750: Procedures {
1751:   SCAN {
1752:     C { SI=\r2 0; SO=\r2 X; }
1753:     // If the STIL consumer's environment allows for a merged scan operation
1754:     If (MergedScan()) {
1755:       Shift { V { SI=#; SO=#; CLK=P; } }
1756:     }
1757:     // Else...STIL consumer's environment requires an unmerged scan operation that includes a
    BreakPoint
1758:     Else {
1759:       Shift { V { SO=#; CLK=P; } }           // unload
1760:       BreakPoint;                          // break patterns here
1761:       Shift { V { SI=#; CLK=P; } }         // load
1762:     }
1763:   } // end scan procedure
1764: } // end Procedures block
1765:

```

Finally, this SCAN procedure is used within the Pattern block to apply all scan patterns. The actual patterns have no special coding depending on whether the merged or the unmerged scan patterns will be used. It is completely encapsulated within the defined SCAN procedure.

```

1766: Pattern SCANPAT {
1767:   Call SCAN { SI1=00100110; SI2=100111; }           // load
1768:   V { CLKS=PPP; PI=10101000001; PO=ZHXXXLHZZXX; }
1769:   Call SCAN { SI1=01000111; SI2=001110;
1770:     SO1=HHLHLLXL; SO2=HXXLHL; }                 // load/unload
1771:   V { CLKS=PPP; PI=11110101000; PO=ZLLLLLHHZZXL; }
1772:
1773:   // ...more patterns not shown
1774:
1775:   Call SCAN { SI1=11110010; SI2=111100;
1776:     SO1=LLLLHXLH; SO2=LHLLLL; }                 // load/unload
1777:   V { CLKS=PPP; PI=11111001001; PO=HLHLLLLLXZL; }
1778:   Call SCAN { SO1=XLHLHLLL; SO2=HHLXLH; }         // unload
1779: }

```

## **M.2 Processing of STIL scan patterns that use the MergedScan() function**

The example in M.1 shows how STIL patterns can be written to allow representation of merged and unmerged scan patterns within the same pattern instance. This clause describes how a STIL consumer may actually process these patterns.

The MergedScan() function is a boolean function that returns true or false based on whether, when processing scan patterns, the merged scan pattern format should be used. By default, this function returns true. It would cause the default behavior of all STIL consumers to output using merged format. During processing of patterns in some given STIL consumer environment, it is determined by the STIL consumer that a need exists to break the patterns because of various factors relevant to that STIL consumer (e.g., Tester Resources, Memory Limitations,...). At this point, the STIL consumer switches the value returned by MergedScan() to be that of false.

The result of the MergedScan() function now returning false will cause the next procedure/macro call that contains a call to MergedScan() to take the alternative path and generate patterns in unmerged format. As part of the STIL consumer making the call to MergedScan() that returns false, the STIL consumer can then reset the MergedScan() function back to its default of state of returning true for all subsequent calls.

In reality, a STIL consumer may not know when it needs to break the patterns up into a new pattern segment until it has already processed to the point in the patterns where resources have been exhausted. In this case, the STIL consumer may need to buffer up STIL patterns, and then when it determines it needs to break the patterns, it will break them at that point and then reprocess the buffered up patterns starting the new pattern segment at the beginning of the most recent procedure call to a procedure containing an If/Else construct based on MergedScan(). It would prevent the STIL consumer from having to know ahead of time, going into a scan procedure, whether the vectors from that procedure will cause an overflow of the resources available.

## Annex N

(informative)

### Labels and X statements for diagnostic feedback

For the purpose of communicating sufficient information to ATE environments so that they can return most meaningful information back to the EDA environment for the purposes of diagnosis, a recommended strategy for using labels and X statements has been developed.

First, the concept of a “test unit” will be introduced. A test unit is a portion of a pattern that represents an extractable, nearly fully contained test. A common example is a scan test composed of a load, one or more capture cycles, and an unload process. Second, the concept of the boundaries of a test unit will be defined. A test unit is composed of stimulus and response across one or more vectors. Given the example of a scan test, the stimulus begins with the load operation and continues into the capture cycles. The response begins in the capture cycles and continues until the end of the unload sequence.

Scan tests may be, and usually are, merged, which means that stimulus from one test unit may overlap response from another test unit. A problem this recommendation is attempting to solve is to permit the unambiguous identification of where a test unit begins and to accurately identify the test unit to which compares belong. Because this overlap occurs, only identifying where a test unit begins, as is commonly done today, makes it impossible to correctly identifying the test unit to which comparisons, and therefore failures, belong.

As a result, it is recommended that labels should be used to identify the first pattern statement of a test unit and that X statements should be used to identify the first pattern statement that might contain comparisons belonging to the test-unit. X statements may also be used to identify other key points in the test unit. It is especially advantageous to take advantage of the hierarchical construction of X statement identifiers by marking the beginning of procedures or macrodefs. An example follows:

```

1780: STIL 1.0 { Design 2005; }
1781: Header {
1782:   Source "STD 1450.1-2005";
1783:   Ann {* annex N *}
1784: }
1785:
1786: Signals { P01 In; P02 In; P03 In; P04 Out; P05 Out; }
1787:
1788: SignalGroups {
1789:   si = 'P01' {ScanIn 4;}
1790:   so = 'P04' {ScanOut 4;}
1791:   PI = 'P01+P02+P03';
1792:   PO = 'P04+P05'; }
1793:
1794: Timing { WaveformTable SIMPLE {
1795:   Period '100ns';
1796:   Waveforms {
1797:     PI {
1798:       01 { '0ns' D/U; }
1799:       P { '50ns' U; '70ns' D; }
1800:     }
1801:     PO {
1802:       LHX { '0ns' Z; '90ns' LHX; }

```

```
1803:    }
1804:  } // end Waveforms
1805: }} // end Timing
1806: Procedures {
1807:   LOAD_UNLOAD {
1808:     W SIMPLE;
1809:     X "Load_Unload";
1810:     Shift { V { P01=#; P04=#; P02=1; P03=P; P05=X; } }
1811:   }
1812:   CAPTURE {
1813:     W SIMPLE;
1814:     X "Capture";
1815:     V { PI=###; PO=###; }
1816:   }
1817:
1818: /* The following is a more complicated "load/unload" that involve pre and post shift vector
statements. In this case the first X statement marks the first first vector of the unload and the second X
statement marks the end of the unload. The trailing X statement is optional if one assumes all vectors are
within a procedure or macro are part of the unload. But the trailing X statement is required if the lines are in
the main pattern block following the call to LOAD_UNLOAD_COMPLEX. */
1819:
1820:   LOAD_UNLOAD_COMPLEX {
1821:     W SIMPLE;
1822:     X "Load_Unload_begin";
1823:     V { PO4=#; }
1824:     Shift { V { P01=#; P04=#; P02=1; P03=P; P05=X; } }
1825:     V { PO4=X; }
1826:     V { PO4=X; }
1827:     V { PO4=H; }
1828:     X "Load_Unload_End";
1829:   }
1830: }
1831: PatternExec "PE_samplepat" {
1832:   PatternBurst "PB_samplepat";
1833: }
1834: PatternBurst "PB_samplepat" {
1835:   PatList { "samplepat"; }
1836: }
1837: Pattern "samplepat" {
1838:   W SIMPLE;
1839:   PATTERN0: Call LOAD_UNLOAD { si=0011; so=XXXX; }
1840:   X "Pattern0"; Call CAPTURE { PI=000; PO=HL; }
1841:   PATTERN1: Call LOAD_UNLOAD { si=0101; so=LLHH; }
1842:   X "Pattern1"; Call CAPTURE { PI=10P; PO=LL; }
1843:   PATTERN2: Call LOAD_UNLOAD { si=1000; so=LHLL; }
1844:   X "Pattern2"; Call CAPTURE { PI=11P; PO=LH; }
1845:   Call LOAD_UNLOAD { si=0000; so=HHHL; }
1846: }
1847:
1848: // Using this example, if the second 'L' and the first 'H' in the pattern statement
1849: // "PATTERN1: call LOAD_UNLOAD { si=0101; so=LLHH; }"
1850: // resulted in a miscompare, the fail data report would appear as follows:
1851:
1852: PatternFailReport {
```

```
1853: Pattern "samplepat";
1854: PatternBurst "PB_samplepat";
1855: PatternExec "PE_samplepat";
1856: FailData {
1857:   "Pattern0"."Load_Unload" P04 1 2;
1858: }
1859: }
```

## Annex O

(informative)

### Use of STIL.1 for specific applications

To be compliant with the STIL.1 standard, a user should apply all parts of the standard that make sense to the application. Parts of the standard may not be applicable to a specific usage, and therefore, not all parts of STIL.1 will be used or usable in all applications (Table O.1). The identification of used/unused parts of the standard is the responsibility of the application. See Clause 1 for the overview of these features/applications.

**Table O.1—STIL.1 statements for specific applications**

Statement in STIL.1	Environment mapping	Parameterized data	Complex test protocol	Advanced scan architecture	Runtime pattern decisions	PatternBurst options	Enhanced user extensibility	Signal relationships	Fail feedback	Usage by ATE <sup>a</sup>	Usage for sub-blocks <sup>b</sup>	Usage for simulation
Variables— IntegerConstant, Integer, SignalVariable, WFCConstant		X	X		X					X <sup>c</sup>	X	X
Signals, SignalGroups— WFCMap								X		X		X
PatternBurst—Variables, Fixed, PatList, PatSet, ParallelPatList			X		X	X				X	X	X
Timing—Variables (domain)										X	X	X
ScanStructures				X							X	X
ScanStructures—cell groups											X	X
pattern-data—\readback- function					X					X	X	X

**Table O.1—STIL.1 statements for specific applications (continued)**

Statement in STIL.1	Environment mapping	Parameterized data	Complex test protocol	Advanced scan architecture	Runtime pattern decisions	PatternBurst options	Enhanced user extensibility	Signal relationships	Fail feedback	Usage by ATE <sup>a</sup>	Usage for sub-blocks <sup>b</sup>	Usage for simulation
Pattern—If, Else, While, Fixed, Equivalent, LoopData, ActiveScanChains, AllowInterleave, BreakPoint, X			X		X			X <sup>d</sup>	X <sup>e</sup>	X	X	X
Pragma										X <sup>f</sup>		
PatternFailReport									X	X		
UserKeywords							X			X	X	X
Environment	X										X	X
Expressions		X	X							X	X	X

<sup>a</sup>It is anticipated that any given ATE system may not be able to fully implement all constructs possible in the STIL.1 extension. In that case, it is expected that the ATE load process or a pre-process application will make necessary adjustments to the file to make it fit the needs of the ATE system.

<sup>b</sup>Much of the additional syntax in STIL.1 is used for the definition of embedded cores (i.e., sub-blocks of a design). These STIL.1 statements are important in the context of (1) defining reusable patterns that are used by SoC patterns, or b) defining embedded cores. Refer to the standard that defines embedded core specification (STIL.6) for the specifics of the usage of these constructs. Note that this information is given only to indicate that the statements listed are important to this application. A given file may or may not use any of these constructs.

<sup>c</sup>ATE usage to only allow Integer variables that include the statement “Usage Test;”.

<sup>d</sup>The Equivalent statement defines signal relationships within a pattern.

<sup>e</sup>ATE software generates PatternFailReport by referencing the X statements in the pattern.

<sup>f</sup>The Pragma must identify the specific ATE system for which it is intended (all others should ignore).

## Annex P

(informative)

### Bibliography

- [B1] IEEE Std 100™, The Authoritative Dictionary of IEEE Standards Terms.<sup>7, 8</sup>
- [B2] IEEE Std 1364-2001, IEEE Standard Verilog Hardware Description Language.
- [B3] IEEE Std 1450-1999, IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data.
- [B4] IEEE P1450.3, Draft Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450-1999) for Tester Target Specification.<sup>9</sup>
- [B5] IEEE Std 1450.6-2005, IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data-Core Test Language (CTL).
- [B6] IEEE Std 1500-2005, IEEE Standard Testability Method for Embedded Core-Based Integrated Circuits.

---

<sup>7</sup>IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (<http://standards.ieee.org/>).

<sup>8</sup>The IEEE standards or products referred to in this annex are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

<sup>9</sup>Numbers preceded by P are IEEE authorized standards projects that were not approved by the IEEE-SA Standards Board at the time this publication went to press. For information about obtaining drafts, contact the IEEE.