IEEE STANDARDS ASSOCIATION

IEEE Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450™-1999) for Test Flow Specification

IEEE Computer Society

Sponsored by the
Test Technology Standards Committee

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

IEEE Std 1450.4™-2017

IEEE Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450-1999) for Test Flow Specification

Sponsor

Test Technology Standards Committee
of the
IEEE Computer Society

Approved 6 December 2017

IEEE-SA Standards Board

Abstract: IEEE Std 1450™-1999, which specifies the Standard Test Interface Language (STIL), is extended by this standard to provide an interface between test generation tools and test equipment with regard to the specification of the flow of execution of test program components. It defines structures so that test flows, sub-flows, and binning may be described in a manner that facilitates automated generation, modification, and/or manual maintenance and, although not yet a complete run-time test language, execution on automated test equipment (ATE). It also defines an interface between tester configurations (described by IEEE Std 1450-1999 and IEEE Std 1450.2™-2002) and test program components. It also defines a hierarchy of flows, sub-flows, and test components as well as structures for defining flow-related variables and processing expressions involving those variables. It provides structures that support automatic test program generation (ATPRG) and translation and that support running it natively as an ATE programming language. As an adjunct, IEEE Std 1450.3™-2007 may be used by ATPRG for tester rules checking.

Keywords: ATPG, ATPRG, automatic test program generator or generation, binning, CAE, computer-aided engineering, device under test, DUT, IC test, IEEE 1450.4™, integrated circuit test, test flow, test program description, test program language, TPG

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

PDF: ISBN 978-1-5044-4643-3 STD22970
Print: ISBN 978-1-5044-4644-0 STDPD22970

Important Notices and Disclaimers Concerning IEEE Standards Documents

IEEE documents are made available for use subject to important notices and legal disclaimers. These notices and disclaimers, or a reference to this page, appear in all standards and may be found under the heading "Important Notices and Disclaimers Concerning IEEE Standards Documents." They can also be obtained on request from IEEE or viewed at http://standards.ieee.org/IPR/disclaimers.html.

Notice and Disclaimer of Liability Concerning the Use of IEEE Standards Documents

IEEE Standards documents (standards, recommended practices, and guides), both full-use and trial-use, are developed within IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association ("IEEE-SA") Standards Board. IEEE ("the Institute") develops its standards through a consensus development process, approved by the American National Standards Institute ("ANSI"), which brings together volunteers representing varied viewpoints and interests to achieve the final product. IEEE Standards are documents developed through scientific, academic, and industry-based technical working groups. Volunteers in IEEE working groups are not necessarily members of the Institute and participate without compensation from IEEE. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, 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.

IEEE Standards do not guarantee or ensure safety, security, health, or environmental protection, or ensure against interference with or from other devices or networks. Implementers and users of IEEE Standards documents are responsible for determining and complying with all appropriate safety, security, environmental, health, and interference protection practices and all applicable laws and regulations.

IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims all warranties (express, implied and statutory) not included in this or any other document relating to the standard, including, but not limited to, the warranties of: merchantability; fitness for a particular purpose; non-infringement; and quality, accuracy, effectiveness, currency, or completeness of material. In addition, IEEE disclaims any and all conditions relating to results; and workmanlike effort. IEEE standards documents are supplied "AS IS" and "WITH ALL FAULTS."

Use of an IEEE standard is wholly voluntary. The existence of an IEEE standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard.

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document, should rely upon his or her own 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.

IN NO EVENT SHALL IEEE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE PUBLICATION, USE OF, OR RELIANCE UPON ANY STANDARD, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND REGARDLESS OF WHETHER SUCH DAMAGE WAS FORESEEABLE.

Translations

The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE should be considered the approved IEEE standard.

Official statements

A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered or inferred to be the official position of IEEE or any of its committees and shall not be considered to be, or be relied upon as, a formal position of 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 of IEEE.

Comments on standards

Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE. However, IEEE does not provide consulting information or advice pertaining to IEEE Standards documents. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important that any responses to comments and questions also receive 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 comments or questions except in those cases where the matter has previously been addressed. For the same reason, IEEE does not respond to interpretation requests. Any person who would like to participate in revisions to an IEEE standard is welcome to join the relevant IEEE working group.

Comments on standards should be submitted to the following address:

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

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so

Copyrights

Photocopies

Subject to payment of the appropriate fee, IEEE will grant users a limited, non-exclusive license to photocopy portions of any individual standard for company or organizational internal use or individual, non-commercial use only. To arrange for payment of licensing fees, 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.

Updating of IEEE Standards documents

Users of IEEE Standards documents should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect.

Every IEEE standard is subjected to review at least every ten years. When a document is more than ten years old and has not undergone a revision process, 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 order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE Xplore at http://ieeexplore.ieee.org/ or contact IEEE at the address listed previously. For more information about the IEEE-SA or IEEE's standards development process, visit the IEEE-SA Website at http://standards.ieee.org.

Errata

Errata, if any, for all IEEE standards can be accessed on the IEEE-SA Website at the following URL: http://standards.ieee.org/findstds/errata/index.html. Users are encouraged to check this URL for errata periodically.

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 by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE-SA Website at http://standards.ieee.org/about/sasb/patcom/patents.html. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Participants

At the time this IEEE standard was completed, the STIL Test Flow Working Group had the following membership:

Jim O'Reilly, *Chair*
Ernst J. Wahl, *Vice Chair*

Gerald Chan*
Kevin Coggins
Julia DiChiaro
Ric Dokken*
Carol Dowding
Dave Dowding
Oleg Erlich
Daniel Fan
Jim Felte
J. Scott Franzen*
Mitsuhito Fujii
Carey Garrenton*
Brian Johnson
Alan Jones
Rohit Kapur
Ajay Khoche
Josie Lewis
Yuhai Ma
Gregory Maston*
Tom Micek
Jim Mosley
Gary Murray
Chris Nelson
Eric Nguyen
Yasunori Okamoto
Don Organ
Bruce Parnas
Paul Reuter
Bob Roberts
Oscar Rodriguez
Jose Santiago
Markus Seuring
Douglas Sprague
Spass Stoiantschewsky
Tony Taylor
S. B. Thum
Steve Tilden
Allen Yeates

(* indicates active membership at the time of draft submission)

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

Paul Berndt
Bill Brown
Juan Carreon
Gerald Chan
Keith Chow
John Cosley
Alfred Crouch
Ric Dokken
David Dowding
Heiko Ehrenberg
Oleg Erlich
J. Scott Franzen
William Fritzsche
Randall Groves
Jon Hagar
Peter Harrod
Werner Hoelzl
Noriyuki Ikeuchi
Gregory Maston
Stephen McGinty
Jeffrey Moore
Charles Ngethe
Jim O'Reilly
Paul Reuter
Osman Sakr
Douglas Sprague
Walter Struppler
Ernst J. Wahl
Yoshihiro Watanabe
Gregg Wilder
Oren Yuen

When the IEEE-SA Standards Board approved this standard on 6 December 2017, it had the following membership:

Jean-Philippe FaureJohn D. Kulick *Chair*
Gary Hoffman, *Vice Chair*
John D. Kulick, *Past Chair*
Konstantinos Karachalios, *Secretary*

Chuck Adams
Masayuki Ariyoshi
Ted Burse
Stephen Dukes
Doug Edwards
J. Travis Griffith
Michael Janezic
Thomas Kochy
Joseph L. Koepfinger*
Kevin Lu
Daleep Mohla
Damir Novosel
Ronald C. Petersen
Annette D. Reilly
Robby Robson
Dorothy Stanley
Adrian Stephens
Mehmet Ulema
Phil Wennblom
Howard Wolfman
Yu Yuan

*Member Emeritus

Introduction

This introduction is not part of IEEE Std 1450.4-2017, IEEE Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450-1999) for Test Flow Specification.

This document is part of a set of IEEE 1450 standards, which cover the Standard Test Interface Language (STIL).

More specifically, this standard (STIL.4) extends IEEE Std 1450™-1999 (STIL.0) to provide an interface between test generation tools and test equipment with regard to the specification of the flow of execution of test program components. It defines

- Structures so that test flows, sub-flows, and binning may be described in a manner that facilitates automated generation, modification, and/or manual maintenance and, although not yet a complete run-time test language, execution on automated test equipment (ATE).
- An interface between tester configurations [described by STIL.0 and IEEE Std 1450.2™-2002 (STIL.2)] and test program components.
- A hierarchy of flows, sub-flows, and test components.
- Structures for defining flow-related variables and processing expressions involving those variables.
- Structures that support automatic test program generation (ATPRG) and translation and that support running it natively as an ATE programming language. As an adjunct, IEEE Std 1450.3™-2007 (STIL.3) may be used by ATPRG for tester rules checking.

Contents

List of Figures

List of Tables

IEEE Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450-1999) for Test Flow Specification

1. Overview

1.1 General

Standard Test Interface Language (STIL) is a standard language that provides an interface between digital test generation tools and test equipment. This standard, referred to as STIL.4, extends IEEE Std 1450™-1999 (STIL.0) to define test flows, enable STIL to tester-language translation, and provide hooks for automatic test program generation (ATPRG).[1]

Test flows direct the execution and sequence of tests. STIL.4 defines TestProgram, Flow, FlowNode, Test, Bin, and FlowVariable blocks to support the definition of test flows. The STIL.4 TestProgram block invokes the test sequence that involves other STIL.4 blocks and references constructs from other STIL standards to create a complete flow. Test operations are defined down to the TestMethod or TestType/Test constructs, which identify invocation but not execution of specific test operations. Figure 1 diagrams the interaction of other STIL standards with STIL.4; when present, a STIL.4 TestProgram identifies the top of the STIL hierarchy.

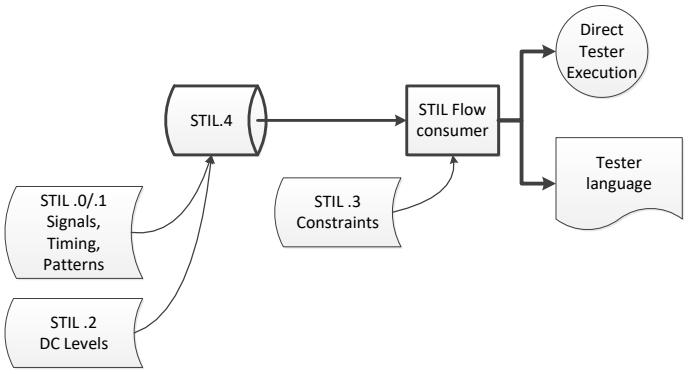

Figure 1— Diagram: STIL flow contents and application

[1] Information on normative references can be found in Clause 2.

STIL.4 supports multiple contexts of use as indicated in Figure 1. Some contexts leverage the ability to use predefined tester interfaces, and the definition of the flow can be specific to that context, in which case the STIL constructs are often directed for this specific context. Other contexts, such as ATPRG usage, require comprehensive and concise semantics in order to translate between tester environments. Not specific to context, STIL.4 identifies two levels of language use, identified with the STIL statement extensions Flow or FlowExtended.

Figure 2 shows a data flow envisioned for ATPRG using STIL. The goal is to, as comprehensively as possible, use STIL as a conduit for automatically generating test programs and retargeting them, i.e., moving them from one tester and/or test environment to another. Retargeting requires special considerations that are not addressed by this standard. Testers X and Y run STIL as the native language. Tester Z runs a proprietary language. Arrows from testers X and Y to ATPRG support incremental test program development.

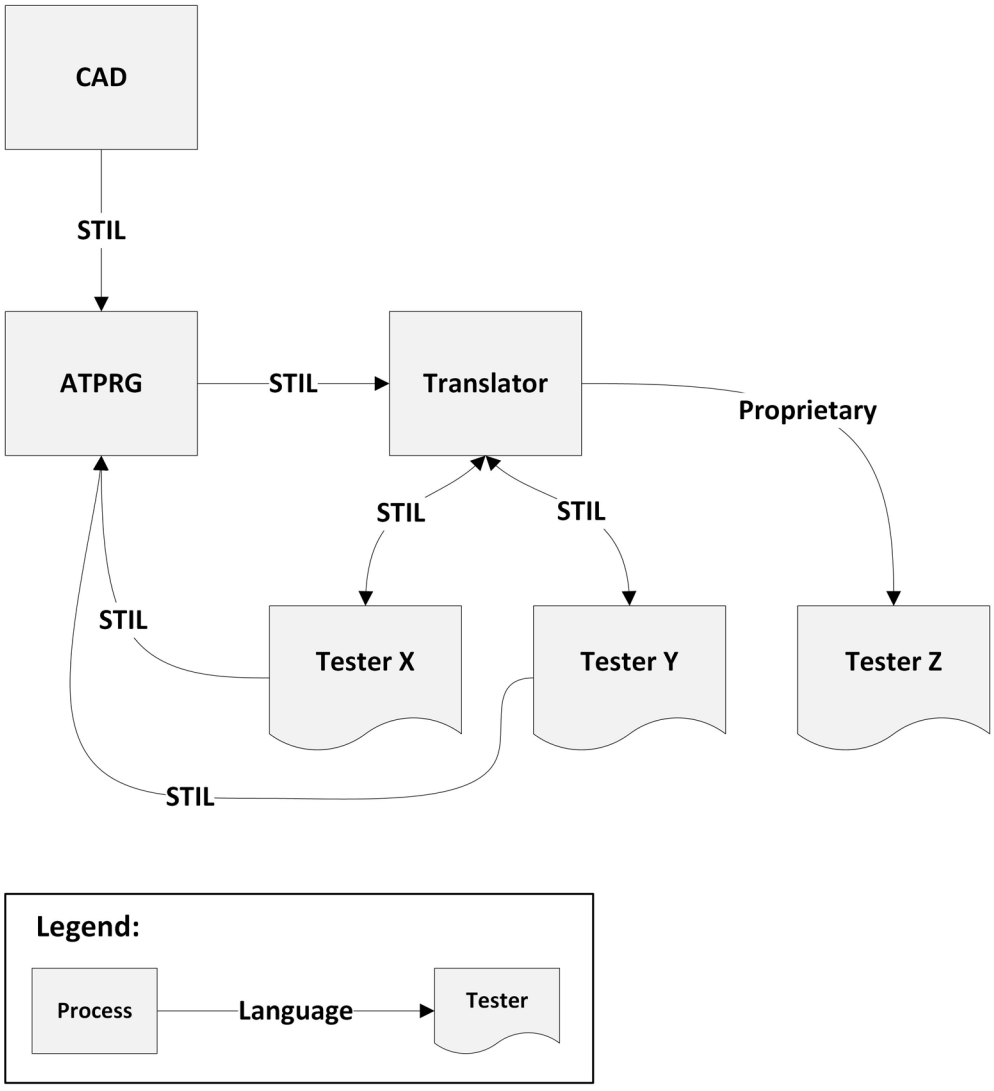

Figure 2—Diagram: ATPRG STIL data flow

1.2 Scope

This standard specifies extensions to STIL.0 that define the description of certain test flow and binning components of an integrated circuit (IC) test program in a test-hardware-independent manner. These extensions provide language constructs and semantics necessary to describe both the test program flow and the sequencing data needed to compose a test program to run on an automated test equipment (ATE) platform. The language constructs defined include structures for specifying the following:

- Order of execution of test program components
- Hierarchical test flow structures to facilitate automated modification or maintenance
- Common interfaces between the test flow environment and test program components
- Test flow variables to facilitate concurrent and serial test flow interactions
- Binning or categorization of tested ICs

The following aspects integral to test execution are specifically not addressed by this standard:

- The standardization of the interface between the prober or handler and tester is beyond the scope of STIL.4. STIL.4 requires that appropriate `AsynchronousEvent` signals shall be issued to the `TestProgram` triggering the corresponding entry-points.
- Input/output operations and exception handling.
- The definition of TestMethods is beyond the scope of this standard.

1.3 Purpose

STIL is the standard for the interchange of digital test data from the test generation environment (where a great deal of design information is used to generate device tests) to the test and manufacturing environment. The initial STIL standard (IEEE Std 1450-1999) addresses the essential digital test description information (i.e., signals, timing, vectors, and parameter specifications). Other aspects needed for testing devices are provided in extension activities such as this standard, which addresses test flow extensions to STIL.

The flow and binning constructs in this extension allow for developing a test program description in a common language; this common description can either be used as input to a test program generator that translates the description into the native language of specific IC ATE systems or be run directly on IC ATE systems that use IEEE 1450.4 as their native language.

2. Normative references

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEEE Std 754™-2008, Standard for Floating-Point Arithmetic. [2,3]

IEEE Std 1450™-1999, IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data.[4]

[2] The IEEE standards or products referred to in Clause 2 are trademarks owned by The Institute of Electrical and Electronics Engineers, Incorporated.
[3] IEEE publications are available from The Institute of Electrical and Electronics Engineers (http://standards.ieee.org/).
[4] This standard combined with IEEE Std 1450.2 and IEEE Std 1450.4 can be used to describe the minimum information required to generate a test program, i.e., timing, levels, patterns, and flow.

IEEE Std 1450.1™-2005, Extensions to STIL for Semiconductor Design Environments.

IEEE Std 1450.2™-2002, Extensions to STIL for for DC Level Specification.[5]

IEEE Std 1450.3™-2007, Extensions to STIL for Tester Target Specification.[6]

3. Definitions, abbreviations, and acronyms

3.1 Definitions

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary Online* should be consulted for terms not defined in this clause.[7]

anonymous: A term to describe an element that has no name/identifier.

assign: To modify the value of an existing instantiation. Value **none** is not legal.

actions: A general term for a syntax subset constrained to specific test, flow, and flow-node sub-blocks. The syntax controls flow via commands that assign, compare, or evaluate variables; set or clear soft bins; and stop program execution.

argument: An expression used to initialize a parameter.

attribute: Ancillary information attached to an object or type describing a property.

buffer: A term to describe a chip interface design cell accessible via a pad.

channel: A physical conduit connecting a tester resource, usually via a tester pin. A channel provides access to, e.g., power, ground, a driver, comparator, control bit.

chip: An instance of a device design on, or cut from, a wafer.

Components: When capitalized, a keyword referring to electrical components such as resistors or capacitors usually mounted on a load-board.

Const: When capitalized, a keyword used as a modifier to denote a variable that represents a single value during normal execution of the test program.

constraint: A special kind of attribute that narrows assignment options.

die: A device design that may be replicated on a wafer in the form of individual chips.

flow or Flow: A general term or, when capitalized, a keyword for a collection of FlowNodes. In FlowExtended mode, a Flow can be viewed as a special kind of test.

flow-node: A general term for a node in a directed graph represented by keyword FlowNode.

[5] This standard combined with IEEE Std 1450 and IEEE Std 1450.4 can be used to describe the minimum information required to generate a test program, i.e., timing, levels, patterns, and flow.
[6] This standard may be used to impose specific target tester limitations on timing, levels, and patterns.
[7] *IEEE Standards Dictionary Online* subscription is available at http://dictionary.ieee.org.

flow-type: A general term for an instantiable flow description.

FlowNode: A keyword for a node in a directed graph.

FlowType: A keyword for a flow-type.

global: A term to describe identifiers visible across an entire test-program. Examples include a variable defined in the unnamed top-level FlowVariables block or a variable defined in a named top-level FlowVariables block that is referenced in the TestProgram block. Variables visible to patterns shall be defined in the unnamed Variables block and require specific attributes.

inheritance: An object-oriented programming reuse mechanism applied to test-types. The new test-type, known as the derived type, inherits attributes and behavior from the pre-existing type, referred to as the base type. Multiple levels of inheritance give rise to an inheritance hierarchy. Only single inheritance is supported, i.e., a test-type can inherit directly only from one other.

initialize: To set starting value(s) at instantiation. Value **none** is legal for variables and optional parameters.

instantiate: To create an instance from a type.

library test-type: A predefined test (flow) type accessible to a Standard Test Interface Language (flow) (STIL.4) test program.

literal: A constant value not represented by a variable, e.g., `1ms` or `"string literal"`.

multi-project wafer: A wafer that has multiple device (chip) types.

multi-site: A term to describe simultaneously testing multiple devices of the same type in parallel.

mutable variable: A variable that does not have type modifier keyword Const and may therefore assume multiple values during the execution of a program.

native library test-type: A library test (flow) type whose TestExec is defined in the native tester language.

none: A sentinel value representing an uninitialized state.

object: An instantiation of a type.

parameter: A typed variable used to pass an argument for object instantiation or on a function call.

partition: A software partition of tester resources (channels).

post-actions: A general term for actions taking place after keyword TestExec in a test (flow) or flow-node including the blocks denoted by keywords PostActions, PassActions, FailActions, and Port.

pre-actions: A general term for actions taking place in the PreActions block of a test or flow-node, i.e., before TestExec.

relational operator: One of operators < <= == != > >=.

resource: A tester resource such as a driver, comparator, parametric measuring unit, switching matrix, etc., accessed via one or more tester channels.

site: A numerical identifier for a device under test (DUT) location.

standard library test-type: A library test-type whose interface and functionality are described in the Standard Test Interface Language (flow) (STIL.4).

target tester: A general term for the tester into whose language the Standard Test Interface Language (STIL) is to be translated.

test: A general term or, when capitalized, a keyword for the smallest executable Standard Test Interface Language (flow) (STIL.4) object.

test-type: A general term for an instantiable test prototype that embodies a definitive procedure that produces a test result.

TestType: A keyword for test-type.

tester: A keyword to identify a target tester. *Syn:* automated test equipment (ATE). *Syn:* **target tester** [with regard to automatic test program generator (ATPRG)].

top-level: A term to describe the definition of a block outside of any other brace enclosed block.

translator: Software that translates the Standard Test Interface Language (vectors and timing/design/flow) (STIL.0/2/4) input to a target tester language.

virtual function: A function whose behavior can be overridden by a derived test-type function of the same name.

3.2 Acronyms and abbreviations

ATE	automated test equipment
ATPRG	automatic test program generator
BNF	Backus-Naur format
DUT	device under test
IC	integrated circuit
LHS	left-hand side
MCM	multi-chip module
MCP	multi-chip package
MPW	multi-project/product wafer
PGA	pin grid array (type of device package)
RHS	right-hand side
STIL	Standard Test Interface Language
STIL.0	IEEE Std 1450-1999 (vectors and timing)
STIL.1	IEEE Std 1450.1-2005 (design)

STIL.2 IEEE Std 1450.2-2002 (levels)

STIL.3 IEEE Std 1450.3-2007 (tester target)

STIL.4 IEEE Std 1450.4-2017 (flow)

4. Preface

4.1 General

STIL.4 is a language with object-oriented features, which describes a test flow designed to be translated into various ATE native-language test flows and provides rudimentary ATPRG support.[8] It may also be used as a fully functional programming language capable of running on ATE.

To generate a test program, an ATPRG or translator input stream is expected to contain STIL.0 code for patterns and timing, STIL.2 code for levels, and STIL.4 code for test flow and ancillary signal information, not necessarily all in one file.

4.2 Word usage

In this document, the word *shall* is used to indicate a mandatory requirement. The word *should* is used to indicate a recommendation. The word *may* is used to indicate a permissible action. The word *can* is used for statements of possibility and capability.

4.3 Conventions

With the exception of the Flow statement (Clause 32) and vector labels, the syntax requires definition before use. In FlowExtended mode, it is recommended that Flows created from FlowTypes be defined before use. In this document, the following conventions are used for syntax definitions:

a) SMALL CAP text for user data
b) **bold** text for keywords
c) *italic* text for meta-types
d) () encloses optional syntax which may be used 0 or 1 time
e) ()+ encloses syntax which may be used 1 or more times
f) ()* encloses optional syntax which may be used 0 or more times
g) <> encloses multiple choice arguments or syntax separated by |

Code examples use `Courier` text. In this document, example lines may be prefaced by line numbers to permit reference for purpose of explanation. Line numbers are not part of STIL.4 syntax. This example illustrates the coding conventions used in this document:

```
1 Enum BlueHues {
2     LIGHT_BLUE,
3     BLUE,
4     DARK_BLUE
5 }
6 TestType MyType {
7     Parameters {
```

[8] See `Signals` *stil4_sig_attrs* in 8.2.

```
 8          In BlueHues blueHues = DARK_BLUE;
 9          In Const Integer Index;
10      }
11      TestExec StdNoOp;
12  }
```

Figure 3—Example: conventions

Here is a more formal description with references to the example in Figure 3:

a) Joined capitalized words are used for the following: constant and type identifiers, and keywords. Note constant identifier Index on line 9, user-defined type identifiers `BlueHues` and `MyType` on lines 1 and 6, and keywords `Enum`, `TestType`, `Parameters`, `In`, `Const`, `Integer`, and `TestExec`.
b) Identifiers that begin with a lower case letter or word potentially followed by joined capitalized words are used for mutable variables and functions. Note identifier `blueHues` on line 8.
c) Enumerations are all capital letters with words separated by underscore. Note lines 2–4.
d) Standard library test-types begin with the letters Std. Note line 11.

4.4 Semantics

This document does not dictate implementation details however on occasion it employs a particular implementation as a vehicle for explaining the actions of and interactions between language elements in concise and unambiguous terms.

STIL.4 compliant code shall be readable by a STIL.4 compliant reader in its entirety. A STIL.4 compliant reader shall read all legal STIL.4 syntax including STIL.4 extensions to STIL.0 and STIL.2. Supported syntax shall be handled in accordance with specified semantics.

5. Tutorial

5.1 General

The following examples describe test programs that may run on an ATE that supports STIL.4 or be directly translated to a native target tester language.

5.2 Flow test program example

The key components of the STIL.4 language extension include Flow, FlowNode, Test, Bin, and FlowVariable blocks. A Flow block may contain a collection of FlowNode blocks. Each FlowNode may execute another Flow or a Test block and then make binning and/or flow navigation decisions at the conclusion of the execution. A Test block defines parameter interface to the execution of the test and may reference a TestMethod. The actual execution of the Test is beyond the scope of this standard.

Figure 4 is a block diagram illustratration of a simple Flow using STIL.4 concepts and terminology. The illustration begins with an ‘On START’ entry point invoking a Flow named MainFlow. MainFlow contains two FlowNodes named dc and ac. The FlowNode dc invokes a second Flow named dcFlow which contains a single FlowNode named iil. The FlowNodes iil and ac invoke Tests. Each of the two Tests in Figure 4 reference a named TestMethod and pass parameter data to the TestMethod. The definition of TestMethod execution and how they use the parameter data is beyond the scope of STIL.4.

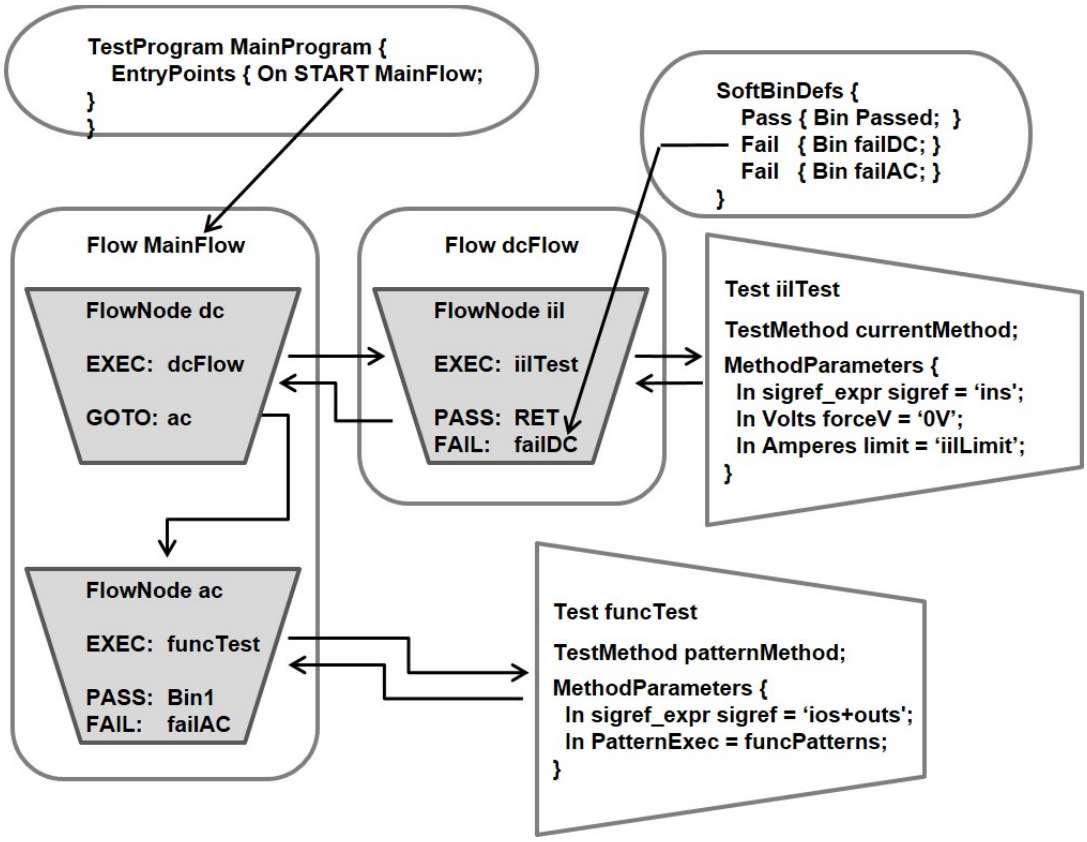

Figure 4—Diagram: STIL flow

Figure 5 shows the corresponding STIL.4 syntax for the example in Figure 4.

```
STIL 1.0 {
  Flow 2017;
  DCLevels 2002;
}
DomainInclude Pat1 { Path "./Patterns/Pat1.stil"; }
FlowVariables {
  Amperes iilLimit='300nA';
}
Signals {
  in0 In;    in1 In;
  io0 InOut; io1 InOut;
  out0 Out;  out1 Out;
  vcore Supply;
}
SignalGroups {
  ins ='in0+in1';
  outs ='out0+out1';
  ios ='io0+io1';
  alldig ='ins+ios+outs';
  pwr ='vcore';
}
SoftBinDefs {
  Pass {
    Bin Bin1 { Number 1; Color "Green"; }
```

```
  }
  Fail {
    Bin failDC { Number 10; Color "Red"; }
    Bin failAC { Number 11; Color "Red"; }
  }
}
Test iilTest {
  TestMethod testCurrent;
  MethodParameters {
    In sigref_expr sigref = 'ins';
    In Volts forceV = '0V';
    In Amperes limit = 'iilLimit';
  }
}
Test funcTest {
  TestMethod testPattern;
  MethodParameters {
    In sigref_expr sigref = 'ios+outs';
    In PatternExec PatExec = Pat1::funcPatterns;
  }
}
Flow MainFlow {
  FlowNode dc {
    TestNumber 1;
    TestExec dcFlow;
    ExitPorts {
      Port 'True' { } Next;
    }
  }
  FlowNode ac {
    TestNumber 2;
    TestExec funcTest;
    ExitPorts {
      Port 'execResult==Pass' { SetBinStop Bin1; }
      Port 'True' { SetBinStop failAC; }
    }
  }
}
Flow dcFlow {
  FlowNode iil {
    TestNumber 11;
    TestExec iilTest;
    ExitPorts {
      Port 'execResult==Pass' { } Return;
      Port 'True' { SetBinStop failDC; }
    }
  }
}
TestProgram {
  EntryPoints {
    On START MainFlow;
  }
}
```

Figure 5—Example: STIL.4 syntax overview

Figure 6 expands Figure 5 by revealing the content within the "./Patterns/Pat1.stil" include file. This include file example defines a pattern execution inclusive of required components. The STIL.4 syntax as shown in Figure 5 simply passes the PatternExec named funcPatterns as a MethodParameter from the Test named funcTest to the TestMethod named patternMethod. How the PatternExec is used by the TestMethod is beyond the scope of STIL.4.

```
STIL 1.0 {
  DCLevels 2002;
}
Signals {
  in0 In;    in1 In;
  io0 InOut; io1 InOut;
  out0 Out;  out1 Out;
  vcore Supply;
}
SignalGroups { all ='in0+in1+io0+io1+out0+out1'; }
DCLevels looseLevels {
  vcore  { VForce '3V'; }
  all    { VIL '0V'; VIH '3V'; VOL '1V'; VOH '2V'; }
}
Timing funcTiming {
  WaveformTable WFT1 {
    Period '100ns';
    Waveforms {
      all { 01LHX { '0s' D/U/Z/Z/Z; '50ns' X/X/L/H/X; } }
    }
  }
}
PatternBurst funcBurst {
  PatList {
    Pat1;
  }
}
PatternExec funcPatterns {
  Timing funcTiming;
  PatternBurst funcBurst;
  DCLevels looseLevels;
}
Pattern Pat1 {
  W WFT1;
  V { all=01XXLH; }
  V { all=01XXLH; }
  V { all=01XXLH; }
}
```

Figure 6—Example: "./Patterns/Pat1.stil" include file

5.3 FlowExtended test program example

The program excerpt in Figure 7 (note the lack of STIL 1.0 statement) performs a single functional test and sets the appropriate fail or pass soft bin. The bin-map contains no soft to hard bin mapping so no hard bin is set. Line numbers are for reference only, not part of the syntax.

```
 1 Signals {
 2   VDD Supply;
```

```
 3   GND Ground;
 4   A0 In; A1 In; A2 In; A3 In;
 5   B0 Out; B1 Out; B2 Out; B3 Out;
 6 }
 7 SignalGroups {
 8   INPUTS  = 'A3 + A2 + A1 + A0';
 9   OUTPUTS = 'B3 + B2 + B1 + B0';
10   ALL     = 'INPUTS + OUTPUTS';
11 }
12 Timing acloose {
13   WaveformTable wft {
14     Period '50ns';
15     Waveforms {
16       INPUTS {
17         01 { '0ns' ForceDown/ForceUp; }
18       }
19       OUTPUTS {
20         HLZ {
21           '0ns' ForceOff;
22           '25ns' CompareHigh/CompareLow/CompareOff;
23         }
24       }
25     }
26   }
27 }
28 DCLevels dcloose {
29   VDD {
30     VForce '3.3V';
31     IClamp '50mA';
32   }
33   INPUTS {
34     VIH '2.0V';
35     VIL '0.8V';
36   }
37   OUTPUTS {
38     VOH '2.7V';
39     VOL '0.4V';
40     IOH '-10mA';
41     IOL '10mA';
42     LoadVRef '1.5V';
43   }
44 }
45 PatternBurst burst1 {
46   PatList {
47     pat1;
48   }
49 }
50 SoftBinDefs softbindefs {
51   Pass {
52     Bin Passed;
53   }
54   Fail {
55     Bin LooseFunct;
56   }
57 }
58 BinMap binmap {
59   SoftBinDefs softbindefs;
```

```
60 }
61 TestProgram basic {
62   BinMap binmap;
63   EntryPoints {
64     On START StdFunctional {
65       failBin  = LooseFunct;
66       passBin  = Passed;
67       patburst = burst1;
68       tim      = acloose;
69       dclev    = dcloose;
70     }
71   }
72 }
73 Pattern pat1 {
74   WaveformTable wft;
75   V { ALL=0000LLLL; }
76   V { ALL=1000HLLL; }
77   V { ALL=0100LHLL; }
78   V { ALL=0010LLHL; }
79   V { ALL=0001LLLH; }
80 }
```

Figure 7—Example: FlowExtended test program

6. Extensions to STIL.0 Clause 6 (STIL syntax description)

6.1 General

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

- Additional STIL reserved words specific within the context of this standard
- Additions to Table 3—SI units
- Extensions to 6.6 (token length)
- Extensions to 6.8 (user-defined name characteristics)
- Extensions to 6.12 (number characteristics)
- Extensions to 6.16 (STIL name spaces and name resolution)

6.2 Additional reserved words

Table 1, Table 2, and Table 3 list all STIL global and context-sensitive reserved words defined by this standard. Subsequent clauses in this standard identify the use and context of each of these additional reserved words. The keywords shown in those two tables shall not be used as identifiers in any context.

STIL.4 identifiers shall use *simple_identifier* syntax and rules described in 6.5. Some identifiers commonly start with integers so that having to quote them would be a burden. Exceptions to the default are indicated in this document where they apply and use *alnum_id* syntax. STIL.0 and STIL.2 identifiers use STIL.0 syntax and rules which is described as *name_segment* also described in 6.5. STIL.4 references to STIL.0 and STIL.2 objects by name shall recognize *name_segment* syntax and rules.

Additionally, `TestType` and `FlowType` names shall not be used as identifiers for any objects. See Clause 7 for an explanation of the Flow and FlowExtended keywords.

Table 1—Additional global STIL.4 reserved words

Else
CurrentExec
execResult
Fail
False
Global
If
Local
None
Parent, Pass
TestNumber
True
Unnamed
While

Table 2—Additional STIL.4 reserved words—Flow

Amperes, AsynchronousEvent
Base, BinAxes, BinAxis, BinMap, BinNumberIncrement, Bins, BinSpec, Boolean, Bypass
Categories, Channel, ClearBin, Color, Const
DCLevels, DCSequence, DCSets, Description, Disconnect, Duration
Enable, EndOfBurst, EndOfPattern, EntryPoints, Enum, ExecResult, ExitPorts
FailActions, Farads, Flow, FlowNode, FlowVariables
General
HardBinDefs, Henries, Hertz
In, Include, IncludeOnce, InOut, Integer
Limits, LotEnd, LotStart
Map, MapBinHighest, MapBinLowest, Max, Meas, Meters, MethodParameters, Min
Next, Number
Ohms, Out
Parameters, PassActions, Permissions, Port, PostActions, PreActions
ReadOnly, ReadWrite, Real, Return, RhsReadWrite
Seconds, Selector, SetBin, SetBinStop, SignalMap, sigref_expr, SoftBinDefs, Spec, SpecVariable, StartBinNumber, Stop, String, Supply
Terse, Test, TestExec, TestMethod, TestProgram, Timing, Typ
Variable, VecLocation, Verbose, Volts
WaferEnd, WafermapChar, WaferStart, Watts, Window

Table 3—Additional STIL.4 reserved words—FlowExtended

ADCtrl, Analog, AnalogDigital
BinGroup, Buffer
Capacitor, ChanDirection, ChannelMap, CheckResult, Chip, CktType, Components, Config, Connect, Control, Coords, Ctap
Design, Device, DeviceSites, DiffNeg, DiffPos, Digital, Diode, DPDT, DPST, DUTBoard, Dynamic
Environment
FailMode, FlowType, ForceHi, ForceLo, Function
Ground, GroundRails
IIH, IIL, Inductor, Inherit, InLevelGrp, IOCtrl, IOType, IOH, IOL, IOZH, IOZL
Layout, Level, LevelGrp, Loadboard, LocType
NC, NO
Off, On, Open, OutLevelGrp
Package, Pad, Pads, Partition, PinList, PinMap, Plane, Power, PowerRails, Probecard, PullDown, PullUp
Rating, Requirement, Resistor
SignalGroup, SigType, Simulation, SPDT, SPST, StaticType, Switch
TestBase, Tester, TestHead, TestMode, TestType, Tolerance, Tristate, Type
Value, Vbreak, Vdrop, VIH, VIHD, VIL, VILD, VOH, VOHD, VOL, VOLD
Wire

6.3 Additions to STIL.0 Table 3 (SI units)

STIL.0 Table 3 is amended to include the units shown in Table 4. Degrees Celsius can be represented as either Cel or oC. Ohms can be represented by Ohm or R.

Table 4—Additions to STIL.0 Table 3

Unit	Description
oC	Degrees Celsius
dB	Decibels
deg	Phase shift or angle
units_expr	General
R	Ohms
none	Real

units_expr contains either simple or combinatorial SI units. For an explanation of combinatorial SI units, see Clause 13.

6.4 Extensions to STIL.0 6.6 (token length)

Tokens are defined to be the block of text between reserved characters, or reserved characters themselves (other than whitespace and comment delimiters). STIL.4 imposes no constraints on token length however STIL.0/1/2 tokens are limited to a maximum length of 1024 characters.

STIL.0/1/2 character strings composed of longer sequences may be defined by segmenting the character string into clauses and placing a period between the clauses.

Although not necessary for consideration of token length, note that STIL.4 strings are concatenated via the plus (+) operator. STIL.4 identifiers shall not be concatenated. The period (.) operator is used in STIL.4 to separate elements in a hierarchical reference.

6.5 Extensions to STIL.0 6.8 (user-defined name characteristics)

The following Backus-Naur format (BNF) forms are added to the forms shown in STIL.0 6.8:

double_quote ::= “**"**”
alnum_id ::= *simple_characters* | *double_quote escaped_characters double_quote*

NOTE—This document uses metatypes *simple_identifier* and *name_segment*, as defined in STIL.0.

6.6 Extensions to STIL.0 6.12 (number characteristics)

The following BNF forms are added to the forms shown in STIL.0 6.12:

number is extended to include the following forms:
signed_integer “e” *signed_integer* “.” *integer* |
signed_integer “.” *integer* e *signed_integer* “.” *integer*

pos_int ::= <**1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**> (*digit*)*****

6.7 Extensions to STIL.0 6.16 (STIL name spaces and name resolution)

BinMap, Chip, Device, FlowVariables, Package, SignalMap, SoftBinDefs, HardBinDefs, Spec, TestProgram, TestType, FlowType, Test, and Flow blocks augment the STIL name space as defined in Table 5. This table is incremental to STIL.0 Table 6 and IEEE Std 1450.2-2002 (STIL.2) Table 2. All definitions present in those two tables remain unchanged.

Table 5—Namespaces

STIL block	Type of name	Domain restrictions
BinMap	BinMap names	Domain names are required and shall be unique across all BinMap blocks.
Chip	Chip names	Domain names are required and shall be unique across all Chip blocks.
Device	Device names	Domain names are required and shall be unique across all Device blocks.
FlowVariables	STIL.4 variable block names	Supports a single unnamed top-level block and domain name (restricted) blocks.
Package	Package names	Domain names are required and shall be unique across all Package blocks.
SignalMap	Signal-to-channel mapping names	Supports a single unnamed top-level block and domain name (restricted) blocks.
SoftBinDefs/HardBinDefs	Bin definition names	Each SoftBinDefs or HardBinDefs block defines an entity. The domain name is required and shall be unique.
TestMethod	TestMethod names	Domain names are required and shall be unique across all TestMethod blocks.
TestProgram	TestProgram names	Domain names are required and shall be unique across all TestProgram blocks.
TestType/FlowType/Test/Flow	Test/Flow type and instance names	Each TestType, FlowType, Test, or Flow block defines an entity. The domain name is required and the name space of all of these entities is shared.

6.8 Expressions

6.8.1 General

STIL.4 supports mathematical, Boolean, and string expressions. An expression may be as simple as a literal value or include variables, operators, and function calls. As the elements within the expression change, the value of the expression changes. For example, the value of expression `'1 + var'` changes whenever the value of `var` changes. The function `eval` evaluates the expression to return a literal value. Single-quoted and unquoted expressions are semantically identical. An expression, whether used directly or represented by a variable, parameter, or spec variable, shall be evaluated whenever a literal value is required. See Figure 8 and Figure 9 for expression assignment and evaluation for Flow and FlowExtended, respectively.

```
 1  FlowVariables {
 2    Integer var = 0;           // Initializes var to 0
 3    Integer a = 1 + var;       // Initializes a to expression 1+var
 4    Integer b = '1 + var';     // Same as above – quotes or not
 5    Integer c = 'eval(1+var)'; // Initializes c to 1
 6    Integer e = 0;             // Initialize e to 0
 7  }
 8  Flow MAINFLOW {
 9    FlowNode {
10     TestNumber 1;
11     PreActions {
12       e=eval(a);  // Evaluates a (1 + var) - sets e to value 1
13       var=2;      // expressions using var will be affected.
14       e=b;        // e is expression (1 + var)
```

```
15       e=e+1;      // Evaluates e, then increments;  e = 4
16       e=eval(a);  // assign value of a; e = 3
18       e=c;        // e = 1
18     }
19   }
20 }
```

Figure 8— Flow expression assignment and evaluation

```
 1 FlowVariables {
 2         Integer var = 0;           // Initializes to value 0
 3         Integer a = 1 + var;       // Initializes to expression
 4         Integer b = '1 + var';     // Initializes to expression
 5   Const Integer c = 1 + var;       // Initializes to value 1
 6         Integer e = eval(1 + var); // Initializes to value 1
 7 }
 8
 9  TestType MyType {                // FlowExtended
10   FlowVariables {
11     Integer c = var;  // Hides global c - local c is set to 'var',
12                       // which will be evaluated when a value is
13                       // needed (i.e., when the PreActions of a Test
14                       // created from MyType are executed).
15     Boolean f = eval(var == 0);  // Initialize to True
16  }
17   PreActions {
18     // Evaluations occur when needed during execution
19     // of PreActions - that is, when a Test created
20     // from this TestType is executed.
21     e = eval(c); // Assigns value 1
22     var = 2;     // Assigns value 2
23     e = b;       // Assigns expression b ('1+var') to e
24     e = e + 1;   // Assigns value 4, not expr e+1.
25     e = eval(c); // Assigns value 2
26   }
27 }
```

Figure 9— FlowExtended expression assignment and evaluation

6.8.2 Literal values

Literal values may be used in variable assignments or passed as parameters. The literal value type shall be compatible with the variable or parameter it is assigned to. To be compatible, the literal value shall either be of the same type as or convertible to the type of variable or parameter it is assigned to (see Table 11). All literal values are constant.

Numeral `3`, for example, is of type `Integer`. The same number with a decimal point, i.e., `3.0`, is of type `Real`, a floating point number with no units. `25e-9` is of type `Real`. Any number with units is of a type associated with those units, e.g., `3s` is of type `Seconds`, a floating point number with units. When the units are combinatorial and not reducible to a simple type, e.g., `3nm:s2` (3 nanometers per second squared), the type is `General`. When units are reducible to a simple type, they are of that type, e.g., `1.0A2R` (1 Ampere squared times Ohms) is of type `Watts`. See Table 4 for a listing of basic unit types, STIL.0 Table 4 for engineering unit prefixes, and the function `units()` description in Table 14 for additional information.

Characters surrounded by double quotes, e.g., `"silly putty"`, are of type `String` unless they've been defined as quoted identifiers in the context of other STIL extension blocks and used in a STIL.4 context that requires a type other than `String`. Double quotes, backslash, `t` for tab and `n` for newline shall be escaped via backslash to be part of a string, e.g., string `"\"C:\\Program Files\"\t\n"` represents windows path name `C:\Program Files` enclosed in quotes followed by tab and newline. It is recommended that quoted identifiers not be used in STIL.0 and STIL.2 code when combined with STIL.4 code. Testers do not in general, support quoted identifiers. There is no separate type or literal that represents a single character.

Keywords `True` and `False` are of type Boolean.

Keyword `None` signifies the value held by an uninitialized variable or parameter. `None` is also the result of illegal or unresolvable operations such as dividing by zero or evaluating an expression containing an undefined variable. While `None` may be a valid default initialization value, it may not be not be explicitly assigned to a variable or passed as a parameter value. A variable or parameter may be tested for equality or inequality to `None`.

6.8.3 Integer expressions (*int_expr*)

Integer expressions consist of integers, spec variables, flow variables, functions (see Table 6), and operators (see Table 13). In FlowExtended, integer expressions may also include test or flow parameters.

An integer expression containing None or a variable that represents None, directly or indirectly, shall evaluate to None.

Integer expressions may include run time variant components such as flow variables. Thus, the resolved value of the expression shall change as the values of the components change. The eval function may be used to force evaluation of an expression and return a literal value.

Examples:

```
1                    // Integer literal
-2+3                 // Integer literal with operator
max(2,50)            // evaluation of integer function
risetime             // Variable
(3 + risetime)/2     // Mathematical operations
```

6.8.4 Real expressions (*real_expr*)

Real expressions consist of integers, real numbers, exponential numbers, spec variables, flow variables, functions (see Table 6), and operators (see Table 13). In FlowExtended, real expressions may also include test or flow parameters. Exponential numbers may be expressed using real numbers or exponential numbers, followed by engineering notation using the prefixes (see STIL.0 Table 4) and units (see STIL.0 Table 3 and, in this document, Table 4).

A real expression containing None or a variable that represents None, directly or indirectly, shall evaluate to None.

Real expressions may include run time variant components such as flow variables. Thus, the resolved value of the expression shall change as the values of the components change. The eval function may be used to force evaluation of an expression and return a literal value.

Examples:

```
1                    // Integer literal
2.0                  // Floating point literal
3ps                  // Floating point literal with units
risetime             // Variable
(3ps + risetime)/2   // Mathematical operations
```

6.8.5 Boolean expressions (*bool_expr*)

A Boolean expression may be as simple as a literal or a predefined variable or it may contain operations performed on literals and/or variables, for example:

```
True                                    // Boolean literal
passed                                  // Boolean variable
(3ps + risetime)/2 <= 5ps               // Boolean expression
boolvar1 != None && boolvar2 == True  // Boolean expression
```

A Boolean expression may use operators `&&`, `||`, `!`, ==, `!=`, >, <, >=, <=, `(` , and `)`.

Boolean operators `&&`, `||`, and `!` shall require Boolean operands.

Boolean operators == and `!=` shall require two operands of the same type, either `String`, `Boolean`, `Integer`, or floating point. One or both of the operands may be `None`. Two expressions are equal if they evaluate to the same value.

Boolean operators >, <, >=, or <= shall require mathematical expression operands and evaluate to `True`, `False`, or `None`. (shall evaluate to `None` if one or more of the operands is `None`).

Boolean operators with both an `Integer` and a `Boolean` operand, shall promote the `Integer` to `Boolean` before the operation is performed (zero is `False`, non-zero is `True`).

6.8.6 String expressions (*string_expr*)

A string expression may be as simple as a literal or a predefined variable or it may contain operations performed on literals and/or variables, for example:

```
""                     // String literal: empty string
"Arbitrary string"     // String literal
"\"\t\n\\"             // String literal of supported escaped characters
str                    // String variable, presumably previously defined
"string " + str        // Concatenating String expression
```

STIL.4 string expressions support operator plus (+) which performs concatenation. STIL.0 and STIL.2 portions of the input stream use operator period (.) for string concatenation. Operator period (.) is not a valid concatenation operator for any identifiers used with STIL.4 constructs. However, it is still a valid concatenation operator for any non-STIL.4 constructs.

For the following STIL.4 statement

```
if ("s" == s) Stop;
```

a STIL.4 compliant parser shall first look for identifier "s". That failing, the parser shall interpret `"s"` as a literal of type `String`.

Inside double quotes, the backslash is used as an escape character causing it and the next character to be interpreted as follows:

Escaped character	Translates to
\"	Double Quote
\\	Backslash
\n	Newline (Linefeed)
\t	Horizontal Tab

6.8.7 SignalGroup (FlowExtended)

6.8.7.1 General

`SignalGroup` is a data type used to pass *sigref_expr* statements. The `SignalGroup` data type has two member functions, **size()** and **at()**.

6.8.7.2 SignalGroup functions syntax

The syntax for the `SignalGroup` data type functions is as follows:

```
Integer     SignalGroup.size();
Signal      SignalGroup.at(int_expr);
```

size: Returns an integer representing the number of ordered signals in the resolved expression.

at: Returns the ordered `Signal` in the `SignalGroup` indexed by *int_expr*. The *int_expr* index of 0 returns the first resolved `Signal`. The *int_expr* index of `SignalGroup`**.size()**-1 returns the last resolved `Signal`. `None` is returned if *int_expr* is out of range, i.e. < 0 or >= `SignalGroup`**.size()**.

6.8.7.3 SignalGroup functions example

The example in Figure 10 illustrates usage of the **size()** and **at()** functions.

```
Signals {
  SIG1 In;
  SIG2 In;
  SIG3 In;
  SIG4 In;
  SIG5 In;
}
SignalGroups {
  GRP1    = 'SIG3+SIG2+SIG1';
  GRP2    = 'GRP1+SIG4+SIG5';   // 'SIG3+SIG2+SIG1+SIG4+SIG5'
}
TestType TestType1 {
  Parameters {
    InOut Const SignalGroup grp1 = 'GRP1';
    InOut Const SignalGroup grp2 = 'GRP2';
    In SignalGroup grp3;
    In Signal sig;
```

```
  }
  FlowVariables {
    Const SignalGroup empty = '';
    Integer int1 = grp1.size();     // Assigns 3 to int1
    Integer int2 = grp2.size();     // Assigns 5 to int2
    Integer int3 = empty.size();    // Assigns 0 to int3
  }
}
Test TestType1 test1 {
  grp1 = &GRP1;
  grp2 = &GRP2;
  grp3 = grp1.at(0);            // SIG3
  sig = grp2.at(grp2.size()-1); // SIG5
}
```

Figure 10—Example: SignalGroup functions

6.9 Functions

STIL.4 supports the functions shown in Table 6. All but function `executed` are globally accessible. The return value shall be `None` or an array of `None` if one or more mathematical expression arguments perform illegal operations or contain an undefined variable or `None`, directly or indirectly. Defined but uninitialized variables evaluate to `None`.

Many of the functions in Table 6 and those described for FlowVariables in Table 11 (in 17.5) and Table 14 (in 17.7) accept one or more arguments of type *math_expr*, which is defined in Table 6 as either an *int_expr* or a *real_expr*.

math_expr ::= < *int_expr* | *real_expr* >

Table 6—Utility functions

Function	Return type	Description
abs(*math_expr***)**	If argument *math_expr* evaluates to type `Integer`, `abs` returns type `Integer`. If argument *math_expr* evaluates to a floating point type, `abs` returns the corresponding floating type.	Returns the absolute value of mathematical expression *math_expr* or, if *math_expr* represents an array, an array of absolute values.
eval(*math_expr***)**	Same as function `abs`.	Directive replacing mathematical expression *math_expr* with the result of its evaluation, either an unquoted literal value or `None`. When used in initialization statement, evaluation occurs at parse time. When used in assignments, evaluation occurs at runtime.

(Table continues)

Table 6—Utility functions *(continued)*

Function	Return type	Description
executed()	`Integer`	Returns the number of times `TestExec` has run to completion since asynchronous event START. The actual number returned depends on context, i.e., `CurrentExec.executed()` counts the number of times the `CurrentExec` *reference_stmt*'s `TestExec` has run to completion from the flow-node. `Parent.executed()` counts the parent test's or flow's `TestExec` executions. `Local.executed()`, same as unqualified `executed()`, counts the test's or flow's `TestExec` executions. `Global.executed()` shall be illegal.
max(*math_expr***,** *math_expr***, ...)**	`Integer` if all arguments are of type `Integer`, or a floating point value if one or more arguments are of floating point types.	Returns a single literal maximum value from the mathematical expression variable-length argument list. Includes array elements in comparison if one or more arguments represent an array. All arguments shall have matching units. NOTE—Argument list is not defined in STIL.0 or STIL.1. If `Flow` extension appears in the `STIL` statement, then STIL.0 and STIL.1 parsers shall use this definition.
min(*math_expr***,** *math_expr***, ...)**	Same as function `max`.	Same as `max` but returns a single literal minimum value.
pow(*math_expr***,** *math_expr***)**	Floating point type	Returns the base (1st argument) raised to the power exponent (2nd argument) as a floating point number, with units if appropriate. The exponent shall be unit less. If the base is scalar then the exponent shall be scalar, and the return value shall be a single literal scalar. If base is an array and exponent is scalar, the return value is an array with every element raised to exponent. If base is an array and exponent is an array, the return value is an array with every element raised to the corresponding exponent, i.e., the arrays shall be of the same size.
str2number(*string***,** *fp***)**	`Integer`	Converts as many of the initial characters in *string* as possible to a floating point number, with units if appropriate. No expression evaluation is performed. Returns the converted character count as type `Integer`. For example, string argument `"10us*2"` sets argument *fp* to `10us` and returns `4`. String argument `"X"` returns `0`.

```
 1 FlowVariables {
 2   Integer i       = -1;          // Fodder for examples below
 3   Integer array[] = [ 2, 3, 4 ]; // Fodder for examples below
 4   Integer i0[]    = array + i;         // Set to 'array + i'
 5   Integer i2[]    = eval(i0);          // Set to [ 1, 2, 3 ]
 6   // Next line shows use of max() with a mix of scalar, array args
 7   Integer i3      = max(i, array, -2); // Set to 4
 8   Integer i4      = abs(-2);           // Set to 2
 9   Integer i5      = abs(-2.5);         // Set to 2 (truncate)
10   Real    r0      = abs(-2.5);         // Set to 2.5
11   General gen0    = pow(4, 0.5);       // Set to 2.0 (Real)
12 }
```

Figure 11—Example: mathematical functions

```
 1 FlowVariables {
 2   Boolean preConditionMet = False;
 3 }
 4
 5 TestType Fnc {
 6   Parameters {
 7     Out Boolean execdFnc = False { ReInitAt TEST_ENTRY; }
 8   }
 9   PreActions {
10     If (Parent.executed())
11       Bypass;
12   }
13   PostActions {
14     If (executed())
15       execdFnc = True;
16   }
17 }
18
19 Test Fnc fnc {
20 }
21
22 Flow StdFlow main {
23   TestExec {
24     FlowNode fn1 {
25       PreActions {
26         If (CurrentExec.executed() == 1)
27           preConditionMet = True;
28       }
29       TestExec fnc;
30     }
31   }
32 }
```

Figure 12— Example: context-sensitive function *executed*

6.10 Enum

6.10.1 Enum syntax

Enum ENUM_NAME {
  (*enum_stmt*)+
}
enum_stmt ::= ENUMERATOR_NAME (=INITIALIZER),

Enum: Start of an enum block.

ENUM_NAME: Required name of the enum block. This name applies scope to the enumerator elements within the enum block.

ENUMERATOR_NAME: Required name of the enumerator element. Each enumerator name within the enum block shall be unique.

INITIALIZER: Optional Integer assignment value for the enumerator. If the initializer is omitted, the enumerator takes on the value of the previous enumerator + 1. The first enumerator, if not assigned with an initializer, is assigned the starting value of 0.

6.10.2 Enum example

```
 1 Enum Color {
 2    BLACK,    // 0
 3    RED,      // 1
 4    YELLOW=4, // 4
 5    BLUE      // 5
 6 }
 7 FlowVariables vars {
 8     Color   c2 = RED;                // Legal: assigns Color::RED
 9     Color   c4 = Color::RED;         // Legal
10     Color   c5;                      // Legal: assigns None
11     Integer i2 = Color::BLUE;        // Sets value to 5
12 }
```

Figure 13—Example: Enum FlowVariables

6.11 Parameter, MethodParameter, and FlowVariable types

The following types and metatypes are used for definition of Test Parameters, MethodParameters, and FlowVariable types.

real_var_type ::= **Amperes** | **Celsius** | **Decibels** | **Degrees** | **Farads** | **General** | **Henries** | **Hertz** | **Meters** | **Ohms** | **Real** | **Seconds** | **Volts** | **Watts**
boolean ::= **True** | **False**

The following definition of ***var_type*** applies when using `Flow 2017` (Clause 7)

var_type ::= <
    Boolean |
    Enum |
    Integer |
    String |
    real_var_type
>

The following definition of ***var_type*** applies when using `FlowExtended 2017` (Clause 7)

var_type ::= (**Const**) <
    Boolean |
    Enum |
    Integer |
    Limits |
    String |
    VecLocation |
    VecRange |
    Window |
    real_var_type
>

The following definitions of *method_param_type and stil_object_expr* are used when defining `MethodParameters` (29.2)

method_param_type ::= <
    var_type |
    DCLevels |
    DCSequence |
    DCSets |
    Pattern |
    PatternBurst |
    PatternExec |
    Signal |
    sigref_expr |
    Timing
>

stil_object_expr ::= anything that evaluates to a *method_param_type*

The following definitions of *param_type* and *stil_object_expr* are used when defining `Parameters` (27.1)

param_type ::= (**Const**) <
    var_type |
    BinSpec |
    Category |
    DCLevels |
    DCSequence |
    DCSets |
    PatternBurst |
    PatternExec |
    Selector |
    Signal |
    SignalGroup | // Covers *sigref_expr*
    Spec |
    SpecVariable |
    TestType |
    Timing
>

stil_object_expr ::= anything that evaluates to a *param_type*

7. Extensions to STIL.0 Clause 8 (STIL statement)

7.1 General

To target intended use cases, this standard has been divided into a basic and extended set of language syntax definitions. The extended set is a superset of the basic set.

Usage of the basic capability set is designated by the `Flow 2017` keyword in the STIL statement block. The basic set imposes minimal changes to usage of clauses defined in existing STIL standards.

Usage of the extended capability set is designated by the `FlowExtended 2017` keyword in the STIL statement block. The extended capability includes additional syntax/semantic changes to clauses in existing

STIL standards. Usage of these features may not be typical in the scope of a test program deployed at a tester. Instead, these features are more likely to be used in offline test program generation applications.

Table 7 shows the STIL.4 clauses are available in both modes (`Flow`) or only in the extended mode (`FlowExtended`). In the remainder of the document, the extended clauses include (`FlowExtended`) in the clause header.

Table 7 — STIL.4 clauses by capability (Flow or FlowExtended)

Clause	Clause title	Flow or FlowExtended
6	Extensions to STIL.0 Clause 6 (STIL syntax description)	Flow (6.2–6.7, 6.8.1–6.8.6, 6.9–6.11) FlowExtended (6.8.7)
7	Extensions to STIL.0 Clause 8 (STIL statement)	Flow
8	Extensions to STIL.0 Clause 14 (Signals block)	FlowExtended
9	Extensions to STIL.0 Clause 15 (SignalGroups block)	FlowExtended
10	Extensions to STIL.0 Clause 16 (PatternExec block)	FlowExtended
11	Extensions to STIL.0 Clause 17 (PatternBurst block)	FlowExtended
12	Extensions to STIL.0 Clause 18 (Timing and WaveformTable block)	FlowExtended
13	Extensions to STIL.0 Clause 19 (Spec and Selector blocks)	Flow
14	Extensions to STIL.2 Clause 10 (DCLevels block)	FlowExtended
15	Extensions to STIL.2 Clause 12 (DCSequence)	FlowExtended
16	Include enhancements	Flow
17	FlowVariables	Flow
18	Device to tester interface	Flow
19	SignalMap	Flow
20	Device	FlowExtended
21	Binning	Flow
22	SoftBinDefs	Flow (22.1–22.3, 22.5) FlowExtended (22.4, 22.6)
23	HardBinDefs	Flow
24	BinMap	Flow
25	Flow conceptual model	Flow
26	Flow conceptual model	FlowExtended
27	TestBase definition	FlowExtended
28	TestType definition	FlowExtended
29	Test	Flow
30	FlowNode	Flow
31	FlowType definition	FlowExtended
32	Flow	Flow
33	Actions and flow control	Flow
34	TestProgram	Flow
35	Standard definitions	Flow (35.1.2) FlowExtended (35.1.3, 35.2–35.7)

7.2 STIL syntax

A STIL statement of the following form is required at the top of each input file:[9]

STIL <*STIL.0_version*> {
  (< **Flow** | **FlowExtended** > <*STIL.4_version*>;)
}

7.3 STIL example

Keyword `Flow` refers to the standard being described in this document. For example, for a file containing STIL.0, STIL.2, and STIL.4 basic syntax:

```
STIL 1.0 {
    Flow 2017;
    DCLevels 2002;
}
```

Keyword `FlowExtended` refers to the standard being described in this document. For example, for a file containing STIL.0, STIL.2, and STIL.4 extended syntax:

```
STIL 1.0 {
    FlowExtended 2017;
    DCLevels 2002;
}
```

8. Extensions to STIL.0 Clause 14 (Signals block) (FlowExtended)

8.1 General

The STIL.4 `Signals` block syntax has the same general outline as STIL.0. STIL.4 supports an optional name for the `Signals` block to support multi-project/product wafer (MPW) and mutli-chip package (MCP) testing. For a single device, including multi-site testing, the unnamed `Signals` block is sufficient. STIL.4 allows additional attributes represented by meta-types *sig_type_stmt* and *stil4_sig_attrs*.

8.2 Signals block syntax and examples

Signals (SIGNALS_NAME) {
  (SIG_NAME *sig_type_stmt*)*
  (SIG_NAME *sig_type_stmt* {
    (**Termination** < **TerminateHigh** | **TerminateLow** | **TerminateOff** | **TerminateUnknown** > ;)
    (**DefaultState** < **U** | **D** | **Z** | **ForceUp** | **ForceDown** | **ForceOff** > ;)
    (**Base** < **Hex** | **Dec** > *waveform_char_list*;)
    (**Alignment** <**MSB** | **LSB**> ;)
    (**ScanIn** (DECIMAL_INTEGER) ;)
    (**ScanOut** (DECIMAL_INTEGER) ;)

[9] Potentially following the `IncludeOnce` statement (see Clause 16).

```
    ( DataBitCount DECIMAL_INTEGER; )
    (stil4_sig_attrs)[10]
  } )*
}
```

Signals: this keyword introduces a brace enclosed block used to define individual signal names and properties. The optional SIGNALS_NAME shall comply with meta-type *name_segment* rules (see 6.5) with one exception: STIL.4 shall not allow SIGNALS_NAME `None`, a keyword used to explicitly denote uninitialized test parameters. Zero or more `Signals` blocks with domain names are allowed. All domain names shall be unique across all `Signals`. A `Signals` block and `SignalGroups` block may have the same name.

sig_type ::= < **In** | **Out** | **InOut** | **Supply** | **Ground** | **Pseudo** >

sig_subtype ::= < **Analog** | **Digital** | **AnalogDigital** | **Open** >

sig_type_stmt ::= *sig_type* (+ *sig_subtype*) (+**Channel**)

Meta-type *sig_type_stmt* provides options that do not exist under STIL.0 and slightly different interpretations for existing STIL.0 options. It is syntactically compatible with STIL.0 syntax. Explanations for *sig_type_stmt* elements are presented in alphabetical order:

Analog: analog signal. Augments signal types `In`, `Out`, `InOut`, `Supply`, or `Ground`.

AnalogDigital: dynamically switchable between analog and digital signal. Augments signal types `In`, `Out`, or `InOut`.

Channel: a non-device-under-test (non-DUT) signal representing a tester channel, which may control loadboard components, e.g., relays, or an alternate tester channel to be connected to a device under test (DUT) signal via relay. Augments signal types `In`, `Out`, `InOut`, `Supply`, or `Ground` (all but `Pseudo`).

Digital: digital signal. For compatibility with STIL.0, the default specification is `Digital` when `Analog`, `AnalogDigital`, or `Open` is not specified. Augments signal types `In`, `Out`, `InOut`, `Supply`, or `Ground`.

Ground: analog or digital ground, fixed at 0V.

Open: static high impedance, usually of type `InOut`, used to test unused package pins for isolation. Augments signal types `In`, `Out`, or `InOut`.

STIL.4 requires a superset of STIL.0 signal types described in Table 8.

[10] Although STIL.0 allows signal attributes to also be applied to groups in the `SignalGroups` block, *stil4_sig_attrs* shall be applied ONLY to signals in the `Signals` block. For explanation of STIL.0 attributes, please refer to the STIL.0 documentation.

Table 8—Signal type/subtype combinations

Signal type	Signal subtype	Interpretation
In	Digital	Digital input.
	Analog	Analog input.
	AnalogDigital	Input switchable between analog and digital.
	Open	Open package pin wanting a drive capable channel.
Out	Digital	Digital output.
	Analog	Analog output.
	AnalogDigital	Output switchable between analog and digital.
	Open	Open package pin wanting a compare capable channel.
InOut	Digital	Digital signal switchable between input and output.
	Analog	Analog signal switchable between input and output.
	AnalogDigital	Signal switchable between input and output, analog and digital.
	Open	Open package pin wanting a drive and compare capable channel.
Ground	Digital	Digital ground.
	Analog	Analog ground.
Supply	Digital	Digital power supply.
	Analog	Analog power supply.
Pseudo	Digital	Digital Pseudo Signal.

Part of that superset merits special attention in that it produces three conceptual categories of signals:

- This signal category, most closely aligned with the STIL.0 notion of a signal, is associated with the chip interface. It is part of this category in that its *sig_type_stmt* includes neither `Pseudo`, `Open`, or `Channel`. Its name performs a dual function. It appears on one or more pads as a chip signal. In the test environment, it also serves as an alias for a tester channel as specified in the context of a `ChannelMap` (see 20.5). This second function facilitates specifying timing, waveforms, and levels on the tester channel in terms of the signal name.
- This signal category is associated with a package pin that has no connection to the chip. It is of type `Open`. It shall not appear on any pad. Its function is to provide provide a signal name that we can associate with a tester channel in the `ChannelMap` in order to specify timing, waveforms, and levels. Typical usage is to perform an isolation test on the corresponding open package pin.
- This signal category is associated with a tester channel. It is of type `Channel`. It may or may not also be associated with a pad. Its function is to provide a signal name alias for a tester channel. That signal name may then be used to specify timing, waveforms, and levels on that channel. Uses include relay control and associating multiple tester channels with a pin or pad. Signal type `Channel` may be added to all type/subtype combinations except `Pseudo`.

stil4_sig_attrs ::=
(< **Pads** (PAD_COUNT) { < (**Pad** (PAD_NUMBER) *pad_attributes*)***** | (*pad_attributes*)***** > } |
Pad (PAD_NUMBER) *pad_attributes* >)
(**Function** *function_stmt***;**)
(**CktType {** *cktype_stmt* **}**)
(**Requirement** *reqmnt_stmt***;**)
(**Environment** *env_stmt*)

Meta-type *stil4_sig_attrs* can only be applied to signals in the Signals block.

CktType *cktype_stmt*

cktype_stmt ::= (**Tristate**;)
(**Dynamic**;)
(**Config** SIG_NAME (NODE_NAME) (, SIG_NAME (NODE_NAME))*;)*

Config: a signal whose programmable buffer instance properties are configured by one or more `Function Control`, `Function IOCtrl`, or `Function ADCtrl` signals. The configuration is controlled by one or more signals indicated by SIG_NAME which uses *name_segment* syntax. Optional NODE_NAME may be used to identify exceptions to the `Function Control` specification indicating which buffer-type node is controlled by which signal. See Figure 19.

Tristate: pad, output, or output portion of ioput is tri-statable.

Dynamic: circuit output state deteriorates over time.

Environment: a variable size list of literal values that describe the electrical environment this output signal was designed to operate in (as opposed to the environment it is being tested in), e.g.:

env_stmt ::= **Design** *value*(, *value*); (**Simulation** *value*(, *value*);)

Design: this describes the environment this signal was designed to operate in.

Simulation: this describes the simulation environment for this signal, i.e., timing may have been generated with a specific tester environment in mind. If this specification is omitted, the assumption is that simulation was done with `Design` value specs.

Here's an example `Environment` specification:

```
Signame Out { Environment { Design 2pF,50R; Simulation 9pF,50R; } }
```

Function *function_stmt*

function_stmt ::= < (**ADCtrl** (NODE_NAME) (= <**ForceHi**|**ForceLo**>)) |
(**IOCtrl** (NODE_NAME) (= <**ForceHi**|**ForceLo**>)) |
(**Control** (NODE_NAME (, NODE_NAME)* (= <**ForceHi**|**ForceLo**>)) |
(**DiffNeg**) | (**DiffPos**) |
(**Power** VOLTS (+TOL) (–TOL) (CURRENT (ACCURACY)))
>

ADCtrl: control signal that switches `AnalogDigital` buffers between `Analog` and `Digital`. `CktType Config` specifies the controlled signal(s). An ATPRG requires a buffer-type library in order to know whether logic 1 maps to `Analog` or `Digital`. Buffer-type library format and content specification is beyond the scope of STIL.4. NODE_NAME refers to a node on a buffer-type.

Control: this signal controls one or more programmable buffer attributes, e.g., tri-state, voltage levels, or drive current capability. A programmable buffer is one whose `CktType` is `Config`. Control may be one of the following:

— static (tied either high or low) or dynamic (controlled by pattern)
— internal (*sig_type* `Pseudo`) or external (accessible by tester channel).

Control parameter(s) NODE_NAME refer to a node on a buffer-type. NODE_NAME specifies the buffer-type node that controls the configuration of the buffer-type. `CktType Config` offers the opportunity to override control connections on a per buffer instance basis if necessary.

`ForceHi` or `ForceLo` is used to permanently lock the programmable buffer-type into a particular state.

DiffNeg: Negative side of differential pair. Pairing is deduced from buffer instance name.

DiffPos: Positive side of differential pair. Pairing is deduced from buffer instance name.

IOCtrl: control signal that switches `InOut` buffers between `In` and `Out`. `CktType Config` specifies the controlled signal(s).

Power: this optional keyword may only be applied to signals of type `Supply`. Its purpose is to help an ATPRG generate levels or configure power channels. VOLTS shall be a literal nominal value with units of volts. TOL optionally indicates either the positive or negative tolerance in percent of the nominal voltage or absolute values in volts. Optional CURRENT and ACCURACY represent the maximal current draw and the most lenient accuracy to which it may be measured; both shall be literal values with units of amperes. Optional ACCURACY shall be preceded by CURRENT. Examples:

```
Power 3.3V;                    // Minimal specification
Power 3.3V +5% -5%;            // More accurate tolerance values
Power 3.3V 3.47V 3.14V;        // Less accurate tolerance values
Power 3.3V 300mA 20mA;         // CURRENT and 20mA accuracy
Power 3.3V +5% -5% 300mA 20mA;// Complete specification
```

Requirement *reqmnt_stmt*

reqmnt_stmt ::= <
(**Ctap** FARADS) |
(**Level** (VOLTS)) |
(**PullDown** OHMS) |
(**PullUp** OHMS)
>

Ctap: low-voltage differential input signals require center tap capacitor to ground as shown in Figure 14, e.g.:

```
Signame In { Requirement Ctap 0.1uF; }
```

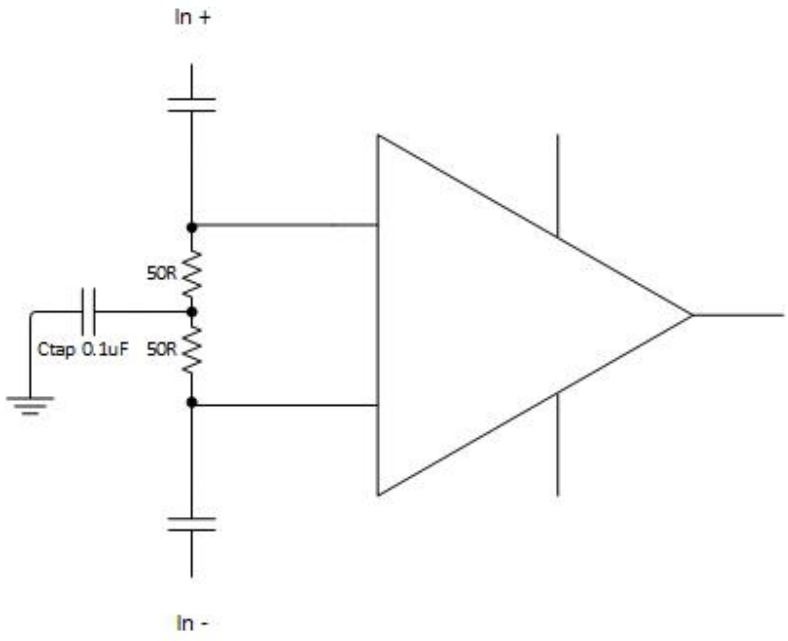

Figure 14—Diagram: LVDS center tap

Level: input requires low current static level such as a reference voltage. Although the following two statements differentiate between analog and digital levels, there is no difference in application other than to indicate a preference for the type of tester resource to use should the ATPRG have the option:

```
Signame In + Analog  { Requirement Level 0.8V; }
Signame In + Digital { Requirement Level 0.8V; }
```

PullDown: output requires pulldown resistor, e.g.:

```
Signame Out { Requirement PullDown 50R; }
```

PullUp: output requires pullup resistor, e.g.:

```
Signame Out { Requirement PullUp 50R; }
```

Pads: this optional keyword associates information about one or more pads with a signal. It is usually applied to signals which appear on multiple pads such as `Supply` or `Ground`. Optional parameter `PAD_COUNT` shall be of meta-type *pos_int*. It describes the number of pads associated with the signal. This keyword shall not be applied to signals of type `Channel`, `Open`, and `Pseudo` (by definition, signals of this type are not associated with pads). For signal type `Open`, connectivity is defined in the `ChannelMap` block, a sub-block of `Device` (see 20.5).

Pad: this optional keyword associates information about one pad with a signal. It provides the opportunity to manually number the pad and associate coordinates and/or buffer information. This keyword shall not be applied to signals of type `Channel`, `Open`, and `Pseudo` (by definition, signals of this type are not associated with a pad). The optional PAD_NUMBER shall be of meta-type *pos_int*. Zero or more tester signals may be associated with that same pad.[11] When multiple channels are associated with a pad, they should act

[11] A chip signal is one whose *sig_type_stmt* includes neither `Pseudo`, `Open`, nor `Channel`. A tester signal is one whose *sig_type_stmt* includes `Channel`.

as a single electrical node either by being ganged or by being constrained to a high impedance (no drive) state. Ganging and high impedance may be specified in `ChannelMap` via the *chan_stmt* plus operator and the *chan_config* `ChanDirection Out` specification respectively (see 20.5).

pad_attributes ::= (**Coords** X_LOC Y_LOC) *buf_stmt*

buf_stmt ::=
< **;** | // Use Semicolon to omit buffer information
< **Buffer** INSTANCE_NAME**;** |
Buffer (INSTANCE_NAME) **{**
< **Type** TYPE_NAME(**.**NODE_NAME); | **Type** (TYPE_NAME(**.**NODE_NAME)) **{** (*buf_attributes*) **}** >
}
>
>

NODE_NAME**:** for single-pad buffer-types, this keyword is optional. For multi-pad buffer-types, it is required. Identifier NODE_NAME selects one of its pads.

Coords: this optional keyword's parameters X_LOC and Y_LOC represent x and y pad center coordinates respectively, specified in units of Meters, e.g., `-3430.22um 3260.39um`. Locations are in relation to an arbitrary origin on the chip die. Usually, there is one pad and therefore one set of coordinates per signal. A signal of type `Supply` or `Ground`, however, may have many pads associated with it and therefore many sets of coordinates per signal. `Coords` may optionally be paired with the `Pad` number. Use `Pads PAD_COUNT` to specify the number of pads when neither pad numbers nor coordinates are available.

Buffer: this optional keyword introduces the buffer instance name. The buffer INSTANCE_NAME shall be required for a multi-pad buffer but is optional for a single pad buffer. INSTANCE_NAME shall conform to meta-type *alnum_id*. When the INSTANCE_NAME is keyword `None`, it signifies an unbuffered pad, i.e., electrical characteristics are unknown.

Type: this optional keyword subordinate to keyword `Buffer` associates the buffer-type name with the signal and may be used to introduce the buffer attributes block. TYPE_NAME shall conform to meta-type *alnum_id*. `NODE_NAME` is optional for a single pad buffer-type. A multi-pad buffer-type shall require `NODE_NAME` to distinguish one pad from another.

buf_attributes ::= (**InLevelGrp** <LEVELGRP_ITEM>**;**)
(**OutLevelGrp** <LEVELGRP_ITEM>**;**)
(**PowerRails** PWR_SIG_NAME (,PWR_SIG_NAME)***;**)
(**GroundRails** GND_SIG_NAME (,GND_SIG_NAME)***;**)
(**IIH** AMPERES (, AMPERES)***;**)
(**IIL** AMPERES (, AMPERES)***;**)
(**IOH** AMPERES (, AMPERES)***;**)
(**IOL** AMPERES (, AMPERES)***;**)
(**IOZH** AMPERES (, AMPERES)***;**)
(**IOZL** AMPERES (, AMPERES)***;**)
(**VIH** VOLTS (, VOLTS)***;**)
(**VIL** VOLTS (, VOLTS)***;**)
(**VOH** VOLTS (, VOLTS)***;**)
(**VOL** VOLTS (, VOLTS)***;**)
(**VIHD** VOLTS (, VOLTS)***;**)
(**VILD** VOLTS (, VOLTS)***;**)
(**VOHD** VOLTS (, VOLTS)***;**)
(**VOLD** VOLTS (, VOLTS)***;**)

buf_attributes: the purpose of buffer-type attributes is to provide values that on a per pad basis, override corresponding values expected to be found in a buffer type description library. These keywords shall be applicable only to signals that are not of type `Supply` or `Ground`. The number of values associated with these keywords shall match the number of `PowerRails` signals. There is positional correspondence so for the following statements:

```
PowerRails VDD5, VDD33;
VOH 2.5V, 1.5V;
```

so for a dual rail programmable buffer, the `VOH` specification is 2.5V when power supply rail `VDD5` is active, the `VOH` specification is 1.5V when power supply rail `VDD33` is active.

GroundRails: specifies the ground signal(s)/rail(s) associated with the buffer. Usually only one is necessary but it is possible that a buffer may switch between analog and digital ground. GND_SIG_NAME is a reference to a previously defined signal of type `Ground`, i.e., it uses *name_segment* syntax. This attribute shall not be applicable to signals of type `Supply` or `Ground`.

InLevelGrp: specifies an input level group to which this buffer instance belongs. LEVELGRP_ITEM refers to an item in the extensible standard-defined `Enum LevelGrp` (35.1). Explicit specification of a specific level (e.g. `VIL` or `VIH`) overrides this implied value.

OutLevelGrp: specifies an output level group to which this buffer instance belongs. LEVELGRP_ITEM refers to an item in the extensible standard-defined Enum `LevelGrp` (35.1). Explicit specification of a specific level (e.g. `VOL` or `VOH`) overrides this implied value. Current capabilities shall be specified separately on a per-buffer or buffer-type basis in a buffer-type library.

PowerRails: specifies the power signal(s)/rail(s) associated with the buffer. Programmable buffers for example may have multiple rails. Only one is expected to be active at any given point in time. PWR_SIG_NAME is a reference to a previously defined signal of type `Supply`, i.e., it uses *name_segment* syntax. This attribute shall not be applied to signals of type `Supply` or `Ground`.

```
 1 Signals CHIP1 {
 2   vdd   Supply + Digital;
 3   vss   Ground + Digital;
 4   vdda  Supply + Analog;
 5   vssa  Ground + Analog;
 6   sig1  In;
 7   sig2  In + Digital;
 8   sig3  In + Analog;
 9   sig4  In + AnalogDigital;
10   sig5  InOut;
11   sig6  InOut + Digital;
12   sig7  InOut + Analog;
13   sig8  InOut + AnalogDigital;
14   sig9  Out;
15   sig10 Out + Digital;
16   sig11 Out + Analog;
17   sig12 Out + AnalogDigital;
18   sig13 Pseudo;
19 }
```

Figure 15—Example: mixed signal Signals block

The examples in Figure 17 and Figure 18 are of a `Signals` block describing the theoretical inverter chip shown in Figure 16.

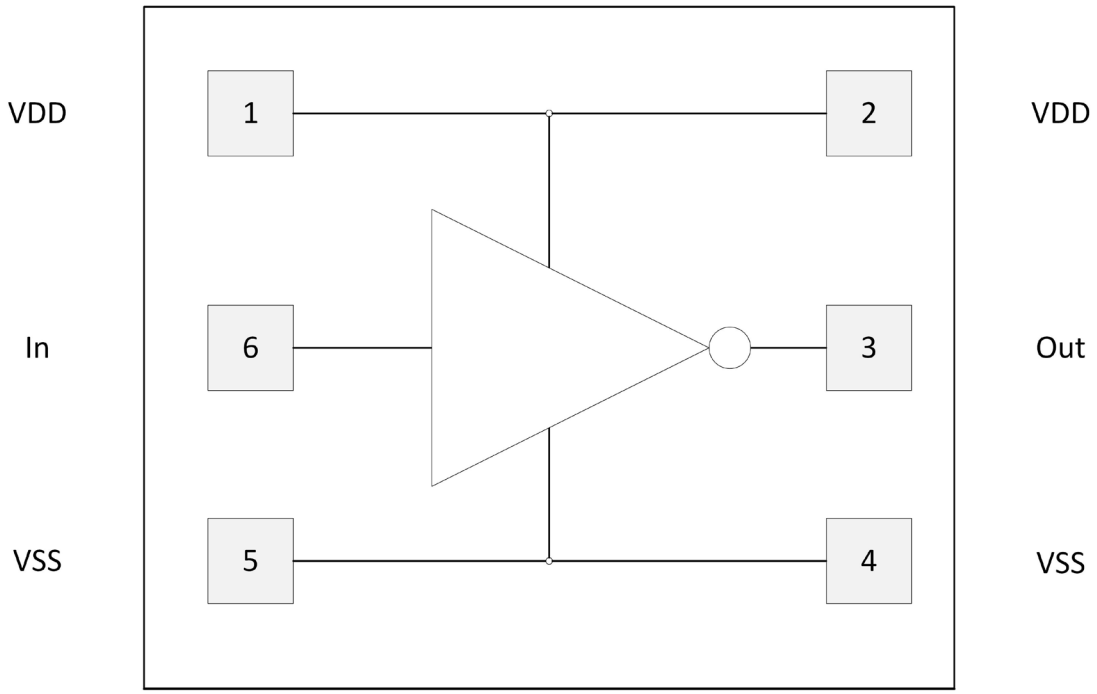

Figure 16—Diagram: inverter chip

The example in Figure 17 includes pad numbering.

```
 1 Signals Inverter {
 2     vdd  Supply {
 3            Pads 2 {
 4                Pad 1 Coords -100um  100um Buffer X1 { Type X; }
 5                Pad 2 Coords  100um  100um Buffer X2 { Type X; }
 6            }
 7     }
 8     vss  Ground {
 9            Pads {
10                Pad 4 Coords  100um -100um Buffer X3 { Type X; }
11                Pad 5 Coords -100um -100um Buffer X4 { Type X; }
12            }
13     }
14     in   In  { Pad 6 Coords -100um   0um Buffer Y3 { Type Y; } }
15     out  Out { Pad 3 Coords  100um   0um Buffer Z3 { Type Z; } }
16 }
```

Figure 17—Example: inverter signals block with pad numbers and coordinates

The following text points out salient syntactical features in the example of Figure 17:

- Line 3: indicates that signal `vdd` is associated with 2 pads. That number may be used to cross-check the number of `Pad` entries in the brace enclosed block.
- Line 4: indicates the pad number, pad coordinates, buffer instance name `X1` and buffer type X. The buffer-type X node name is optionally unspecified indicating that the buffer-type has only one pad, i.e., it would be required for a multi-pad buffer type as, e.g., `X.a` (letter `a` is the node name).
- Line 9: indicates that signal `vss` is associated with the pads listed in the brace enclosed block, i.e., lines 10 and 11. There can be no cross-check on the number of `Pad` entries since the optional pad-count following keyword `Pads` was omitted.

The example in Figure 18 omits pad numbers and coordinates but adds other attributes.

```
 1 Signals Inverter {
 2     vdd  Supply {
 3         Pads 2 {
 4             Buffer X1 { Type X.a; }
 5             Buffer X2 { Type X.a; }
 6         }
 7         Function Power 5V +5% -5% 500mA 20mA;
 8     }
 9     vss  Ground {
10         Pads 2 {
11             Buffer X3 { Type X.a; }
12             Buffer X4 { Type X.a; }
13         }
14     }
15     in   In  { Pad Buffer Y3 { Type Y.a {
16                                    InLevelGrp  TTL;
17     } } }
18     out  Out { Pad Buffer Z3 { Type Z.a {
19                                    OutLevelGrp TTL;
20     } } }
21 }
```

Figure 18—Example: inverter signals block, no pad numbers or coordinates

The following text points out salient syntactical features in the example of Figure 18:

- Line 4: within the `Pads` block the optional keyword `Pad` is not employed hence the pad number remains unspecified. Each line still represents a pad. Unlike in the example of Figure 17, the buffer-type X node-name is specified, i.e., the name is `a`.
- Line 7: specifies a nominal `5V` level, a tolerance of `±5%`, a maximal expected dynamic current draw of `500mA`, and desired measurement accuracy to within `20mA`.
- Line 16: specifies `InLevelGrp`, one of various buffer attributes specified in this document as meta-type *buf_attributes*. This specification applies only to buffer instance `Y3`. A buffer library may be used to look up buffer-type `Y`, node `a`, to find this and other specifications. Buffer attributes specified in this context override those found in a buffer library.

Figure 19 shows three multi-pad programmable buffers whose output characteristics are affected by signals on nodes `a` and `b`.

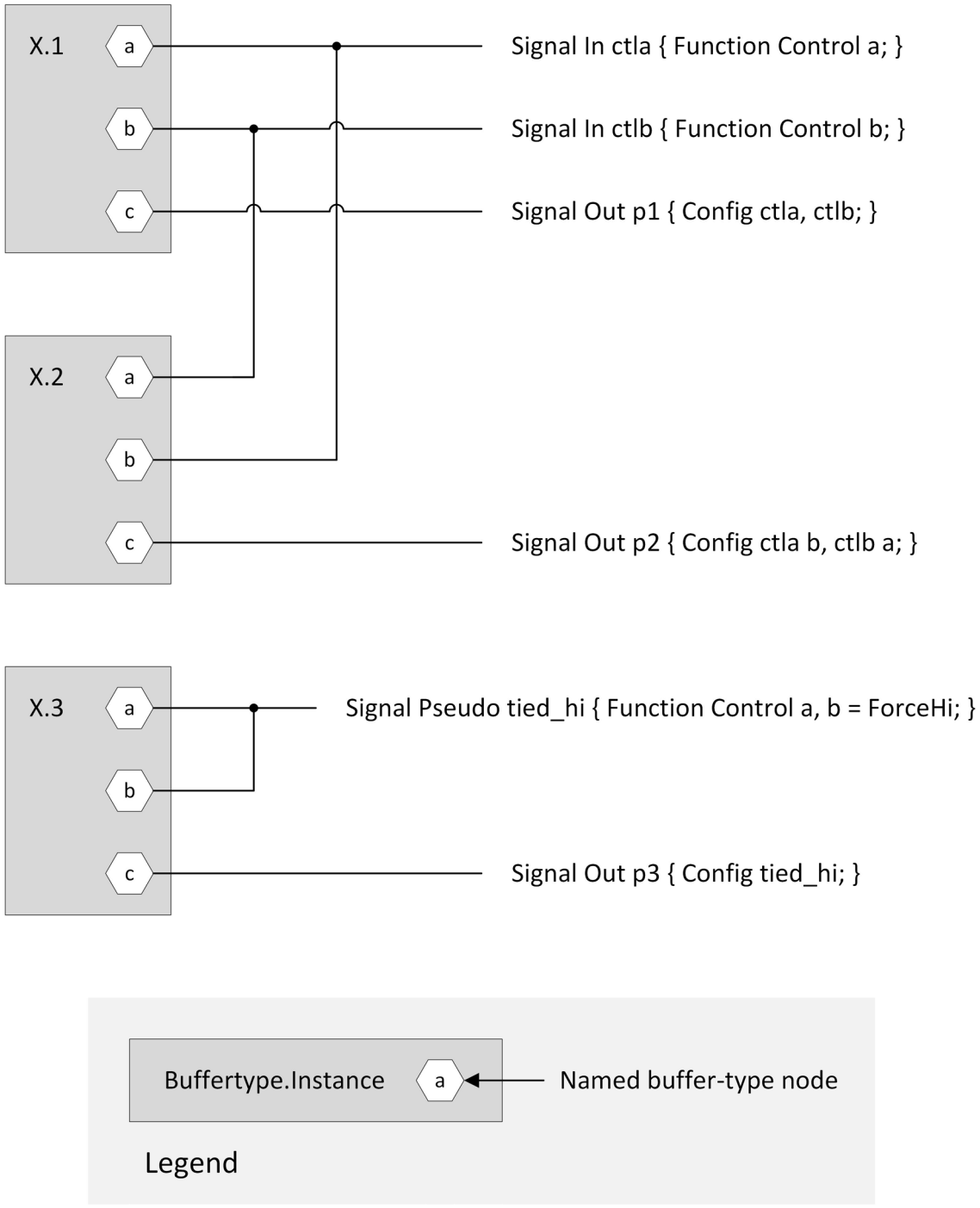

Figure 19—Diagram: programmable buffers

The `Signals` block example in Figure 20 corresponds to Figure 19.

```
 1 Signals CHIP2 {
 2     vdd     Supply;
 3     vss     Ground;
 4     ctla    In     { Function Control a; } // Norm
 5     ctlb    In     { Function Control b; } // Norm
 6     tied_hi Pseudo { Function Control a, b = ForceHi; } // Tied high
 7
 8     p1      Out    {
 9         Config ctla, ctlb;                 // Follows norm
10         Pad { Buffer 1 { Type X.c; } }
11     }
12
13     p2      Out    {
14         Config ctla b, ctlb a;             // Specifies exception
15         Pad { Buffer 2 { Type X.c; } }
16     }
17
18     p3      Out    {
19         Config tied_hi;
20         Pad { Buffer 3 { Type X.c; } }
21     }
22 }
```

Figure 20—Example: programmable buffers

9. Extensions to STIL.0 Clause 15 (SignalGroups block) (FlowExtended)

STIL.4 allows for multiple named top-level `Signals` blocks. The unnamed top-level `Signals` and `SignalGroups` blocks form a single namespace, i.e., no signal may have the same name as a signal-group. Identically named `Signals` and `SignalGroups` blocks also form a single namespace. When in a `SignalGroups` block, a signal-group is defined in terms of one or more references to a signal or signal-group on the right-hand side by name, each reference is resolved in the following order:

a) The current `SignalGroups` block is searched for the name. A match resolves to the references contained in that name.
b) A reference is matched against the names defined in the top-level unnamed `SignalGroups` block. A match resolves to the references contained in that name.
c) A reference is matched against the names defined in the `Signals` block of the same name as the `SignalGroups` block.
d) A reference is matched against the names defined in the top-level unnamed `Signals` block.

When named `Signals` or `SignalGroups` blocks are specified in the `Device/Package` block directly, or `Device/Chip` block indirectly, those named blocks shall in effect augment the top-level unnamed blocks with regard to the name pooling and name resolution mechanisms described above. For consistent behavior between named and unnamed blocks, STIL.4 shall not permit signal names to be overridden by signal group names.

NOTE—STIL.0 and STIL.2 permit references to `SignalGroups` definitions at lower levels which take precedence inside the block where that reference is made. Blocks that permit this are of type `PatList`, `PatternBurst`, `Timing`, `DCLevels`, or `DCSequence`. STIL.4 signal and signal-group resolution lookup order: individual local (e.g., `Timing`) block `SignalGroups`, `Device` block `SignalGroups`, including those brought in via `Chip`, i.e., the named `SignalGroups` and `Signals` blocks and/or unnamed `SignalGroups` and `Signals` blocks.

10. Extensions to STIL.0 Clause 16 (PatternExec block) (FlowExtended)

10.1 General

A `PatternExec` is not executable from within STIL.4 however a user or ATPRG may instantiate `FlowExtended` test-type `StdPatternExec` passing in `PatternExec` as a parameter or `FlowExtended` test-type `StdFunctional` passing in some or all elements of `PatternExec` as parameters.

The `PatternExec` block syntax shown in 10.2 is a composite of STIL.0 and STIL.2 specifications. STIL.4 additionally sanctions specifying `Category` by including `SPEC_BLOCK_NAME`:

10.2 PatternExec block syntax

PatternExec (PAT_EXEC_NAME) {
  (**Category** (SPEC_BLOCK_NAME**.**)CATEGORY_NAME;)*
  (**Selector** SELECTOR_NAME;)*
  (**DCLevels** (DC_LEVELS_NAME);)
  (**DCSets** (DC_SETS_NAME);)
  (**Timing** TIMING_NAME;)
  (**PatternBurst** PAT_BURST_NAME;)
}

11. Extensions to STIL.0 Clause 17 (PatternBurst block) (FlowExtended)

11.1 General

STIL.4 shall accept `PatternBurst` block syntax as defined in STIL.1 Clause 12 without requiring the `Design 2005` statement in the STIL block. As per STIL.0 7.1, references to pattern names are expected to be forward references. These references need to be resolved but not necessarily within STIL.4 code.[12]

11.2 Extensions to STIL.0 17.1 (PatternBurst block syntax)

This clause augments STIL.0 syntax. It allows the user to make device or chip associated signal and signal group definitions visible from within the `Timing`, `DCLevels`, `DCSequence`, and `Pattern` blocks, i.e., a change made at the device or chip level is automatically reflected within all the aforementioned blocks.

PatternBurst PAT_BURST_NAME {
  (**Device** DEVICE_NAME; | **Chip** CHIP_NAME;) // STIL.4: tracks changes made to Device or Chip
  (**SignalGroups** GROUPS_DOMAIN;)* // STIL.0: error prone, tracks neither Device nor Chip
    •
    •
    •
  (**PatList** {

[12] Two separate processes are likely used, one for and patterns, the other for timing, levels, and flow. See Figure 1 where the pattern translator is first process and the ATPRG bundled with the program translator is the second, respectively.

```
    ( PAT_NAME_OR_BURST_NAME ; )*
    ( PAT_NAME_OR_BURST_NAME {
      (Device DEVICE_NAME; | Chip CHIP_NAME;)  // STIL.4: tracks changes made to Device or Chip
      (SignalGroups GROUPS_DOMAIN; )*          // STIL.0: error prone, tracks neither Device nor Chip
        •
        •
        •
  } )*   // End of PAT_NAME_OR_BURST_NAME
 })+  // End of PatList
}  // End of PatternBurst
```

Chip: makes the signal and signal group definitions gathered under the `Chip` block available for subsequent *sigref_expr* statements.

CHIP_NAME: reference to a previously defined `Chip` block.

Device: makes the signal and signal group definitions gathered under the `Device` block available for subsequent *sigref_expr* statements.

DEVICE_NAME: reference to a previously defined `Device` block.

SignalGroups: makes named signal group definitions available for subsequent *sigref_expr* statements. For STIL.4, it is preferable to use `Device` or `Chip` or the unnamed `Signals` and `SignalGroups` blocks. As in STIL.0, signals and signal groups defined in unnamed `Signals` and `SignalGroups` blocks are visible everywhere.

GROUPS_DOMAIN: reference to a previously defined `SignalGroups` block.

STIL.0 timing and pattern lists, and STIL.2 DC levels do not support testing multiple devices in a single environment. Using keyword `Device` or `Chip` instead of `SignalGroups` accomplishes the following:

— Makes signal and signal group definitions gathered under the `Device` or `Chip` block available for subsequent *sigref_expr* statements. Signal definitions include chip signals and potentially signals representing, e.g., unused package pins (isolation tests) and tester channels (loadboard relay control).

Timing, pattern lists, and DC levels track signal and signal group changes made at the `Chip` or `Device` level.

12. Extensions to STIL.0 Clause 18 (Timing and WaveformTable block) (FlowExtended)

12.1 General

This clause augments STIL.0 syntax. It allows the user to make device or chip associated signal and signal group definitions visible from within the `Timing`, `DCLevels`, `DCSequence`, and `Pattern` blocks, i.e., a change made at the device or chip level is automatically reflected within all the aforementioned blocks.

12.2 Timing and WaveformTable syntax

Timing (TIM_DOMAIN_NAME) {
(**Device** DEVICE_NAME; | **Chip** CHIP_NAME;) // STIL.4: tracks changes made to Device or Chip
(**SignalGroups** GROUPS_DOMAIN;)* // STIL.0: error prone, tracks neither Device nor Chip
•
•
•
}

Chip: makes the signal and signal group definitions gathered under the `Chip` block available for subsequent *sigref_expr* statements.

CHIP_NAME: reference to a previously defined `Chip` block.

Device: makes the signal and signal group definitions gathered under the `Device` block available for subsequent *sigref_expr* statements.

DEVICE_NAME: reference to a previously defined `Device` block.

SignalGroups: makes named signal group definitions available for subsequent *sigref_expr* statements. For STIL.4, it is preferable to use `Device` or `Chip` or the unnamed `Signals` and `SignalGroups` blocks. As in STIL.0, signals and signal groups defined in unnamed `Signals` and `SignalGroups` blocks are visible everywhere.

GROUPS_DOMAIN: reference to a previously defined `SignalGroups` block.

STIL.0 timing and pattern lists, and STIL.2 DC levels do not support testing multiple devices in a single environment. Using keyword `Device` or `Chip` instead of `SignalGroups` accomplishes the following:

— makes signal and signal group definitions gathered under the `Device` or `Chip` block available for subsequent *sigref_expr* statements. Signal definitions include chip signals and potentially signals representing, e.g., unused package pins (isolation tests) and tester channels (loadboard relay control).

Timing, pattern litsts, and DC levels track signal and signal group changes made at the `Chip` or `Device` level.

13. Extensions to STIL.0 Clause 19 (Spec and Selector blocks)

13.1 General

Assignment of an initial value to the Meas field of a Spec variable is now explicitly allowed. STIL.0 describes the existence of the Meas field, but states that assignment of a value to the Meas field occurs during test program execution. This is still the case; however, the initial value syntax, together with the ReInitAt statement, permits the assignnment of an initial value to the Meas field and allows re-initialization of the Meas field to that value when the specified ASYNC_EVENT_NAME event occurs.

Spec expressions follow the characteristics defined for timing expressions in STIL.0 6.13 and for DC expressions in STIL.2 5.2.

Spec expressions are enclosed in single quotes and contain the same entities as timing, DC, or real expressions. Expressions that compute to any units shown in IEEE 1450-1999, Table 3, including the additions shown in Table 4, may be used in a spec expression. In addition, complex expressions that are expressed as a ratio of two values (e.g., a slew rate of "1V/1ns") may be used in a spec expression.

Spec expressions occur only as part of a spec variable definition in the Spec block.

Keyword `None` can be used a sentinel value to indicate that a spec variable not initialized.

The colon (`:`) operator can be used to represent combinatorial units; i.e., `3.5V:ns` means "3.5 volts per nanosecond (3.5V/ns)". Further, one can specify an unsigned integer or unitless real number following a units-symbol (both before and after the colon operator) to indicate exponentiation of units, e.g., `1V:s2` means "1 volt per second squared". These extensions shall apply to *time_expr*, *dc_expr*, and *real_expr*.

Syntax rules for combinatorial units: all units, before or after the colon operator, shall have an implicit superscript of 1 or shall be followed by an unsigned integer or real number (a number that includes decimal point) indicating exponentiation of the preceding units. Products of units can be specified by listing the units and exponents in sequence both before and after the colon operator, where positive exponents come before and negative exponents after. See Table 9 for examples of combinatorial units.

Table 9—Example: combinatorial units

Expression	Description
`V:s`	Volts per second, Volts raised to the power of 1 divided by seconds raised to the power of 1, implicit
`V1:s1`	Volts per second, Volts raised to the power of 1 divided by seconds raised to the power of 1, explicit
`V:s2`	Volts per second squared
`:A2`	Amperes raised to the power of -2
`A0.5`	Amperes raised to the power of ½, i.e., the square root
`A2R:mCel`	Watt per meter-degree Celsius (thermal conductivity)

13.2 Spec block syntax

spec_expr ::= *time_expr* | *dc_expr* | *real_expr*

Spec (SPEC_NAME) { // this block statement defines variable values for a given category
(**Category** CAT_NAME {
(VAR_NAME = *spec_expr*;)* | // Defines only Typ value
(VAR_NAME {
((**Min** *spec_expr*;) (**Typ** *spec_expr*;) (**Max** *spec_expr*;) (**Meas** *spec_expr*;)) | (**Units** "*units_expr*";)
(**ReInitAt** ASYNC_EVENT_NAME;)
})*
})+
}

Spec (SPEC _ NAME) {
// this block statement defines category values for a given variable
(**Variable** VAR _ NAME {
(CAT _ NAME = *spec_expr*;)* | // defines only the **Typ** value
(CAT _ NAME {
((**Min** *spec_expr*;) (**Typ** *spec_expr*;) (**Max** *spec_expr*;) (**Meas** *spec_expr*;)) | (**Units** "*units_expr*";)
(**ReInitAt** ASYNC_EVENT_NAME ;)

```
      } )*
   } )+
}
```

The units of any **Min**, **Typ**, **Max**, or **Meas** field values specified shall be the same. If no values are specified for any of the fields or if all that are specified are explicitly initialized to **None**, then **`Units`** *"units_expr"* may be specified in order to provide for error checking of the units of values assigned to the **Meas** field during program execution.

If no units are specified via either method above, then the first assignment to the Meas field that occurs in the program flow shall assign units (including no units).

Specifying **ReInitAt** ASYNC_EVENT_NAME has no effect on any **Min**, **Typ**, or **Max** selector fields, but allows the **Meas** field to be re-initialized upon the occurrence of the specified asynchronous event.

14. Extensions to STIL.2 Clause 10 (DCLevels block) (FlowExtended)

14.1 General

STIL.4 requires STIL.2 to perform functional and related tests.

When parsing STIL.2 input, a STIL.4 compliant parser accepts STIL.4 units, a superset of STIL.0/STIL.2 units (see Table 4).

This clause augments STIL.0 syntax. It allows the user to make device or chip associated signal and signal group definitions visible from within the `Timing`, `DCLevels`, `DCSequence`, and `Pattern` blocks, i.e., a change made at the device or chip level is automatically reflected within all the aforementioned blocks.

14.2 DCLevels block syntax

```
DCLevels (DC_LEVELS_NAME) {
      (Device DEVICE_NAME; | Chip CHIP_NAME;)  // STIL.4: tracks changes made to Device or Chip
      (SignalGroups GROUPS_DOMAIN; )*          // STIL.0: error prone, tracks neither Device nor Chip
         •
         •
         •
}
```

Chip: makes the signal and signal group definitions gathered under the `Chip` block available for subsequent *sigref_expr* statements.

CHIP_NAME: reference to a previously defined `Chip` block.

Device: makes the signal and signal group definitions gathered under the `Device` block available for subsequent *sigref_expr* statements.

DEVICE_NAME: reference to a previously defined `Device` block.

SignalGroups: makes named signal group definitions available for subsequent *sigref_expr* statements. For STIL.4, it is preferable to use `Device` or `Chip` or the unnamed `Signals` and `SignalGroups` blocks.

As in STIL.0, signals and signal groups defined in unnamed `Signals` and `SignalGroups` blocks are visible everywhere.

GROUPS_DOMAIN: reference to a previously defined `SignalGroups` block.

STIL.0 timing and pattern lists, and STIL.2 DC levels do not support testing multiple devices in a single environment. Using keyword `Device` or `Chip` instead of `SignalGroups` accomplishes the following:

— Makes signal and signal group definitions gathered under the `Device` or `Chip` block available for subsequent *sigref_expr* statements. Signal definitions include chip signals and potentially signals representing, e.g., unused package pins (isolation tests) and tester channels (loadboard relay control).

Timing, pattern litsts, and DC levels track signal and signal group changes made at the `Chip` or `Device` level.

15. Extensions to STIL.2 Clause 12 (DCSequence) (FlowExtended)

15.1 General

This clause augments STIL.0 syntax. It allows the user to

— Make device or chip associated signal and signal group definitions visible from within the `Timing`, `DCLevels`, `DCSequence`, and `Pattern` blocks, i.e., a change made at the device or chip level is automatically reflected within all the aforementioned blocks.
— Control, within a DCSequence, the opening or closing of a switch as defined in the Components subblock of the Device block, and the time relative to other events in that DCSequence when the opening or closing occurs.

STIL.4 supports per tester/testhead specific definitions for `DCSequence InitialSetup`, `PowerRaise`, `PowerLower`, `EndOfProgram`, and `User` in addition to top-level namespace definitions specified in STIL.2 Clause 18. For a `DCSequence` defined under the `Device block`, STIL.4 ignores the optional `SignalGroups` statement. This is because the `Device block` already has the relevant signals and signal groups imported into its environment via its `Chip, Signals,` and `SignalGroups` statements. The use of DCSequence blocks under the Device block hides any DCSequence top-level blocks with the same name.

STIL.4 requires STIL.2 to perform functional and related tests.

When parsing STIL.2 input, a STIL.4 compliant parser accepts STIL.4 units, a superset of STIL.0/STIL.2 units (see Table 4).

15.2 DCSequence block syntax

DCSequence (DC_SEQ_NAME) {
    (**Chip** CHIP_NAME;) // STIL.4: tracks changes made to Chip
    (**SignalGroups** GROUPS_DOMAIN;)* // STIL.0: error prone, tracks neither Device nor Chip
    •
    •
    •

time_expr[13] { (**Switch** relay_name < **On** | **Off** >;)+ }
}

Chip: makes the signal and signal group definitions gathered under the `Chip` block available for subsequent *sigref_expr* statements.

CHIP_NAME: reference to a previously defined `Chip` block.

SignalGroups: makes named signal group definitions available for subsequent *sigref_expr* statements. For STIL.4, it is preferable to use the `Chip` block or the unnamed `Signals` and `SignalGroups` blocks. As in STIL.0, signals and signal groups defined in unnamed `Signals` and `SignalGroups` blocks are visible everywhere.

GROUPS_DOMAIN: reference to a previously defined `SignalGroups` block.

STIL.0 timing and pattern lists, and STIL.2 DC levels do not support testing multiple devices in a single environment. Using keyword `Chip` instead of `SignalGroups` accomplishes the following:

— Makes signal and signal group definitions gathered under the `Chip` block available for subsequent *sigref_expr* statements. Signal definitions include chip signals and potentially signals representing, e.g., unused package pins (isolation tests) and tester channels (loadboard relay control).

Timing, pattern lists, and DC levels track signal and signal group changes made at the `Chip` or `Device` level.

Note that the `DCSequence` block only has syntax to track `Chip`. This is because for DC sequences defined at the top level, the `Device` and `DCSequence` blocks would be mutually dependent were keyword `Device` an option inside the `DCSequence` block. When `DCSequence` blocks are defined inside the `Device` block, the device is known eliminating the need for a `Device`, `Chip`, or `SignalGroups` reference to make signal and signal group definitions visible inside the `DCSequence` block. Test-type default parameters or test instantiation parameter may reference top-level DC sequences.

For **Switch** actions, the *sigref_expr* normally present on other lines (as specified in STIL.2) is absent. This is because the loadboard relay is hardwired hence, specifying a signal is meaningless.

15.3 DCSequence block example

For controlling a loadboard relay, STIL.4 recognizes the following format inside the `DCSequence` block:

time_expr[14] { (**Switch** relay_name < **On** | **Off** >;)+ }

Switch off is in the normal default position. Switch on is the other position. For example:

```
1 DCSequence InitialSetup { // Beginning of program
2     '0s' 'VDD+VDDA' {     // T0
3         Apply '0V';
4     }
5     '1ms' 'VDD+VDDA' {    // Wait 1ms, then connect
6         Connect Supply;
7     }
```

[13] This is *time_expr* as used in STIL.2.
[14] This is *time_expr* as used in STIL.2.

```
 8      '1ms' {                // Wait 1ms, then switch
 9          Switch S1 On;      // Closes normally open relay
10      }
11  }
```

16. Include enhancements

16.1 IncludeOnce

IncludeOnce: this is an optional semicolon terminated instruction that shall precede any STIL language statement in the file. If this statement is present in file `FILE_NAME`, and statement `Include "FILE_NAME"` (see STIL.0 10.1) appears multiple times in the input stream, only the first `Include "FILE_NAME"` statement shall be honored, the rest are ignored.

The following example illustrates the use of `IncludeOnce`:

File *tap_timing.stil*:

```
IncludeOnce;     // Precedes any STIL language statement in the file
.
Timing Tap {...}
.
```

File *dc_param_timing.stil*:

```
.
Include "tap_timing.stil";
Timing DC_Param {...}
.
```

File *main.stil*:

```
.
Include "tap_timing.stil";
Include "dc_param_timing.stil";
.
```

File *tap_timing.stil* contains timing blocks for a test access port. File *dc_param_timing.stil* contains timing blocks for setting up DC parametric tests, but includes *tap_timing.stil* for completeness. Assuming that file *main.stil* is opened as the STIL input stream, *tap_timing.stil* is only included once, i.e., the `Include tap_timing.stil` statement in file *dc_param_timing.stil* is ignored during the insert of *dc_param_timing.stil* by *main.stil*.

16.2 DomainInclude

16.2.1 General

A **DomainInclude** statement allows STIL blocks defined in other files to be referenced while avoiding name conflicts.

16.2.2 DomainInclude syntax

The **DomainInclude** statement takes the following form:

```
DomainInclude DOMAIN_NAME {
  Path "FILE_NAME";
}
```

The *domainreference_stmt*, referred to in other parts of this document takes the following form:

domainreference_stmt =
< DOMAIN_NAME::BLOCK_NAME |
DOMAIN_NAME::BLOCK_TYPE >

DOMAIN_NAME is the namespace assigned to the included file.

BLOCK_NAME is the user assigned name to the STIL syntax block within the included file. This can only apply to named STIL syntax blocks.

BLOCK_TYPE is the STIL statement type of the syntax block within the included file. This can only apply to unnamed STIL syntax blocks or to blocks where there is only one object of that type.

The *domainreference_stmt* may be used anywhere in the STIL.4 language that a local file reference to a STIL block can be used.

16.2.3 DomainInclude example

Figure 21 shows an example usage of the **DomainInclude** statement.

```
STIL 1.0 { Flow 2017; DCLevels 2002; }
DomainInclude Pat1 {
  Path "./Patterns/Pat1.stil";
}
Signals {
  in0 In;  in1 In;  io0 InOut;  io1 InOut;  out0 Out;  out1 Out;
  vcore Supply;
}
PatternBurst funcBurst {
  PatList { Pat1::Pattern; }  // reference to one and only Pattern
}
PatternExec Exec1 {
  Timing Pat1::Timing;        // reference to unnamed Timing block
  DCLevels Pat1::looseLevels; // reference to named DCLevels block
  PatternBurst funcBurst;
}
Test funcTest {
  TestMethod testPattern;
  MethodParameters {
    In PatternExec PatExec = Exec1;
  }
}
```

—Example: Main test program file

```
STIL 1.0 {
  DCLevels 2002;
```

```
}
Signals {
  in0 In;  in1 In;  io0 InOut;  io1 InOut;  out0 Out;  out1 Out;
}
SignalGroups { all ='in0+in1+io0+io1+out0+out1'; }
DCLevels looseLevels {
  all    { VIL '0V'; VIH '3V'; VOL '1V'; VOH '2V'; }
}
Timing {
  WaveformTable WFT1 {
    Period '100ns';
    Waveforms { all { 01LHX { '0s' D/U/Z/Z/Z; '50ns' X/X/L/H/X; } } }
  }
}
Pattern func_pat {
  W WFT1;
  V { all=01XXLH; }
  V { all=01XXLH; }
  V { all=01XXLH; }
}
```

—Example: "./Patterns/Pat1.stil" include file

Figure 21 —Example: domainInclude statement

17. FlowVariables

17.1 General

`FlowVariables` blocks contain variables and constants. Typical uses for these variables and constants are to control flow of execution and to provide meaningful names for values.

Only one unnamed `FlowVariables` block may be defined at the top level. All other `FlowVariables` blocks shall be uniquely named. An unnamed `FlowVariables` block may be defined for any object derived from `TestBase`.

`TestProgram` block `FlowVariables` refererences are to named top-level variables blocks. Variables in the referenced blocks are global to the test program. The unnamed variables block is referred to implicitly and shall be read before named blocks which are read in the order specified. It shall be an error for defined variable or constant names to conflict with `Signals`, `SignalGroups`, or `Spec` variable names accessible in the same context.

17.2 FlowVariables syntax

(**FlowVariables** (VAR_DOMAIN) { (**ReInitAt** ASYNC_EVENT_NAME;) *var_elements_stmt** })*

The optional `ReInitAt` statement specifies the default re-initialization event for all variables in this block. ASYNC_EVENT_NAME is an enumeration from type `AsynchronousEvent` defined in 35.1. If this statement is absent, the default initialization event is `START`.

var_elements_stmt ::= <
    var_type var_definition_stmt | // Individual form

var_type { (*var_definition_stmt*)+ } // Block form
>

var_type is defined in 6.11
var_definition_stmt ::= <
// Uninitialized scalar variable, optional type matching *units* allowed for *real_var_type* only
VAR_NAME (= **None**(*units*)); |
VAR_NAME (= **None**(*units*)) { (*var_attributes*)* } |

// Initialized scalar variable.
VAR_NAME = *value_expr*; |
VAR_NAME = *value_expr* { (*var_attributes*)* } |

// Unintialized array variable
// optional type matching *units* allowed for *real_var_type* only
VAR_NAME[*int_expr*] (= **None**(*units*)); |
VAR_NAME[*int_expr*] (= **None**(*units*)) { (*var_attributes*)* } |

// Initialize array elements to distinct values
VAR_NAME[(*int_expr*)] = [*value_list*]; |
VAR_NAME[(*int_expr*)] = [*value_list*] { (*var_attributes*)* } |

// Initialize all array elements to same value
VAR_NAME[*int_expr*] = *value_expr*; |
VAR_NAME[*int_expr*] = *value_expr* { (*var_attributes*)* }
>

The following definition of ***value_expr*** applies when using `Flow 2017` (Clause 7)

value_expr ::= <
int_expr | // Allowed for variable of type `Integer`, `Boolean`, and *real_var_type*
sigref_expr | // Allowed for scalar variable of type `sigref_expr`
bool_expr | // Allowed for variable of type `Boolean`
string_expr | // Allowed for variable of type `String`
real_expr | // Allowed for variable of type `Integer` (truncated), and *real_var_type*
>

The following definition of ***value_expr*** applies when using `FlowExtended 2017` (Clause 7)

value_expr ::= <
int_expr | // Allowed for variable of type `Integer`, `Boolean`, and *real_var_type*
sigref_expr | // Allowed for scalar variable of type `SignalGroup` or `sigref_expr`
bool_expr | // Allowed for variable of type `Boolean`
string_expr | // Allowed for variable of type `String`
real_expr | // Allowed for variable of type `Integer` (truncated), and *real_var_type*
limits | // Allowed for variable of type `Limits`
vecloc | // Allowed for variable of type `VecLocation`
vec_range | // Allowed for variable of type `VecRange`
window // Allowed for variable of type `Window`
>

var_attributes ::= <
Description *string*; // Default is: empty string, i.e. "".
| **ReInitAt** ASYNC_EVENT_NAME;

```
    | Units "units_expr";                  // Useful for type General with initial value of None
    | Permissions <ReadWrite | RhsReadWrite | ReadOnly>;
    | SiteSharePer <Tester | TestHead | Partition>; // Default is: not shared.
    | Usage< Pgm | Test>
>

value_list ::= < value_expr | value_list, value_expr >

var_assignment_stmt ::= <
    VAR_NAME = value_expr; |                    // Scalar variables
    VAR_NAME[int_expr] = value_expr; |  // Array variables
    VAR_NAME = [value_list];                // Array variables
>
```

Variable definition and initialization has individual and block forms. The individual form requires repeating type information for each variable, while the block form extracts the common denominator type information.

Both scalar and array variables may be defined. Arrays may be single-dimensional or multi-dimensional. Multi-dimensional arrays have additional sets of brackets, each representing a dimension, i.e., an array axis. Any mathematical expression inside these brackets, quoted or unquoted, used to define a dimension size shall immediately be evaluated to an integer, 0 or greater, at the instantiation of the array. If the initialization value on the right-hand side is a fully defined array, the sets of brackets on the left-hand side need not contain dimensions. Left- and right-hand side dimensions shall be required to match if both are specified unless the right-hand side is explicitly or implicitly `None`. The type of each array element shall match *type_name*. All array elements shall have identical units[15] even when *type_name* is `General`. Array dimensions and units shall remain fixed once the array is initialized, i.e., assigned a value other than `None`.

Note that initialization and assignment rules and behaviors are the same except that the ability to specify attributes/constraints and the ability to set the value `None` are reserved for initialization only. Figure 23 shows legal initialization code and further clarifies which statements are legal and what the results of these operations are.

When a constant variable is instantiated with an expression containing one or more mutable variables, mutable variables on the right-hand side are replaced with their values at the time of instantiation. Figure 24 illustrates this behavior.

17.3 FlowVariables examples

```
 1 FlowVariables {
 2     String StdChipType = "B2509" {
 3         Permissions RhsReadWrite;
 4         ReInitAt LOAD;
 5         Description "Chip type identifier";
 6     }
 7     Seconds hold = 1ns;
 8 }
 9 FlowVariables globals {
10     Seconds delay = 0s;
11 }
12 TestProgram pgm {
13     FlowVariables globals;
14 }
```

Figure 22—FlowVariables example

[15] Units only apply to floating-point types.

```
 1 Enum Colors { // Defined for use in examples below
 2     NO_COLOR; // 0
 3     RED;      // 1
 4     YELLOW;   // 2
 5     BLUE;     // 3
 6 }
 7 FlowVariables legal {
 8     Integer  i   = -3;
 9     Integer  ir  = 3/2;              // ir = 1; integer división

       // ie = 2; automatic conversion from Enum to Integer conversion.
       // Colors:: qualifier needed since assigning to type than Colors
10     Integer  ie  = Colors::YELLOW;
11     Integer  ir  = 3/2;              // ir = 1; integer division
12     Boolean  b   = True;
13     Boolean  b0  = 0;                // b0 = False

       // c = Colors::RED. Assigning to vartype Colors; hence, no
       // need for Colors:: qualifier on RHS value Red
14     Colors   c   = RED;
15     Integer  i3  = c;               // i3 = 1
16     String   s1;                    // s1 = None
17     String   s3  = "";              // s3 = <empty string> (!= None)
18     String   s4  = "How now brown cow";

       // Variables g1, g2 and, g3 are constrained to units of s after
       // initialization by various means
19     General  g1 { Units "s"; }       // Value of g1 is None
20     General  g2  = None;            // Value of g2 is None
21     General  g3  = 0s;               // Value of g3 is 0s
22     Real     r   = 2;                // Value of r is 2.0
23     Seconds delay = 2ns;

       // The literal Limits value on RHS requires parentheses
       // delimiters
24     Limits lims   = (0s <= .I. <= delay);
25     Const Limits {
           // Uninitialized variable open; default constraints,
           // attributes
26         open;
27         volts       = None { Units "V"; }
28         pos seconds = (0s <= .I. < None);
29     }
30 }
```

Figure 23—Example: scalar variable initialization

When assigning expressions to variables, be they constant or mutable, the intent is to retain as much of the original expression as possible. This informs the user that a value may have several constituent parts and helps with tracing the origin of those parts. This second example illustrates such initialization behavior.

```
 1 FlowVariables {
 2   Seconds prdm   = '10ns';      // Stores 10ns
 3   Seconds prdc   = 20ns;        // Stores 20ns
 4   Seconds strb1  = '0.9*prdm';  // Stores '0.9*prdm' (eval deferred)
 5   Seconds strb3  = eval(strb1); // Stores 9ns (force evaluation)
 6 }
```

Figure 24—Example: scalar variable initialization

```
Integer array[]  = [ 1, 2, 3 ]; // Size 3
Integer array[3] = [ 1, 2, 3 ]; // Size 3, both sides match
Integer array[3] = 0;           // Size 3, elements initialized to 0
Integer array[]  = None;        // Unknown size one dimensional array
Integer array[];                // Same as above
```

Figure 25—Example: array initialization

```
Integer array[2][3] = 0;
Integer array[2][3] = None;
Integer array[2][3];            // Initialized to None, same as above
```

Figure 26—Example: array initialization, all elements set to the same value

```
Integer array[2][3] = [         // Dimensions match right-hand side
    [ 11, 22, 33 ],
    [ 21, 22, 23 ],
];
```

Figure 27—Example: multi-dimensional array, per element initialization

```
Integer i = array[0][0];
array[0][0] = 5;
array = 4;                     // All elements set to 4
```

Figure 28—Example: array element access and assignment

17.4 FlowVariable access

The unnamed top-level `FlowVariables` block is implicitly referenced by the `TestProgram` block hence, a variable in that block may be accessed simply by its name. A variable in a named top-level `FlowVariables` block is accessible by its name only if the `TestProgram` block references that `FlowVariables` block by name. The `TestProgram` block shall not reference variable blocks with conflicting variable names.

A variable defined in a `TestType`'s `FlowVariables` block shall be accessible to that test-type only, i.e., a derived test-type shall not access base-type variables. Each instance of a particular test type shall have its own copy of variables. For code within the scope of a test, a test parameter or variable identifier, e.g., `period`, shall hide a top-level variable of the same name. Operators `Global` and `Local` may be used to select the scope. For example, `Global.period` shall refer to the top-level variable `period` even when a local definition is present. The same applies to `FlowType`.

Global: used to unambiguously refer to a top-level variable, e.g., `Global.period`.

Local: used to indicate and/or insure that a local test parameter or variable is being referred to. For example: after removing a variable, e.g., `period`, from a test-type definition, usage of that variable may unintentionally find top-level `period`. Specifying `Local.period` insures that a compile time error locates the unintentional use.

Note that a flow-node does not have its own scope hence a reference to `Local.period` in a flow-node is a reference to variable or parameter `period` in the test or flow that contains the flow-node.

17.5 FlowVariable types

Variable types represent a data structure, real, integer, Boolean, or string. Specific types may represent a further specialization of one of these classifications. All variable types have identifiers, constraints, attributes, functions, and operators. See 6.5 for constraints on identifier names.

Constraints are either intrinsic or user-settable. Intrinsic constraints are implied by the variable type, e.g., type `Seconds` can only be initialized to or assigned a value of type `Seconds` whereas type `General` may be initialized to any numeric value. User settable constraints may occur in several locations. Type-modifier keyword `Const` occurs before the type-name and constrains a variable or parameter to be immutable. Other constraints, located between braces along with attributes, restrict legal assignments to a subset of what might otherwise be legal.

Variables are defined in blocks preceded by keyword `FlowVariables` which can occur at the global level and/or inside test-type definitions.[16]

During initialization, the right-hand side, i.e., the value, of a variable to be assigned may be a reference to another that has already been initialized.

Table 10 shows the types that represent floating point values and shows the units for each type.

[16] One global variable block may optionally be unnamed. The unnamed global variable block and the global named variable blocks specified in the `TestProgram` block shall be accessible to STIL.4. Variable blocks inside test-type definitions shall be unnamed.

Table 10—Real types

Type name	Units	Measure of …
Amperes	A	Current
Celsius	oC | Cel	Temperature
Decibels	dB	Logarithmic Ratio
Degrees	deg	Phase shift or angle
Farads	F	Capacitance
General	*units_expr*	*various*
Henries	H	Inductance
Hertz	Hz	Frequency
Meters	m	Distance
Ohms	R | Ohm	Resistance
Real	*none*	Magnitude
Seconds	s	Time
Volts	V	Potential
Watts	W	Power

Table 11 shows STIL.4 FlowVariable types and the literal values that may be associated with them.

Table 11—FlowVariable types

Type name	Legal values	Automatic conversion	User-settable constraints	Type-specific member functions
Boolean	`None` | `True` | `False` | *int_expr* | *bool_expr* | ENUMERATOR			
Enum	ENUMERATOR	To `Integer`: Uses ENUMERATOR value To `Boolean`: ENUMERATOR value 0 to `False`, all other values to `True`		size(), string()
Real [a]	See Table 10.	Truncated when assigned to type `Integer`	Type `General`: `Units` "*units_expr*"; All others: implied by type	units()
Integer	`None` | *int_expr*	To `Real`: `None` to `None`, integer to real To `Boolean`: `None` to `None`, 0 to `False`, others to `True`		
String	`None` | "*char**" | *string_expr*	To `Enum`		size()

(Table continues)

Table 11—FlowVariable types *(continued)*

Type name	Legal values	Automatic conversion	User-settable constraints	Type-specific member functions
FlowExtended only				
Limits	`None` | *math_expr*, <=, <		`Units` "*units_expr*";	units(), check(*math_expr*), hi(), lo()
VecLocation	`None`|*vecloc*		*LocType*:`None`| `LABEL`;	locType()
VecRange	`None`|*vecloc*		*LocType*:`None`| `LABEL`;	locType()
Window	`None`|*vecloc*|*sigref_expr*		*LocType*:`None`| `LABEL`;	locType()

[a] Types are enumerated in Table 4: they include `Real`, `Seconds`, `General`, etc.

Automatic single step conversion occurs between types as indicated.

Boolean: a small structure representing a Boolean literal or expression that when evaluated, shall reduce to either `True` or `False` unless it performs an illegal operation or directly or indirectly contains `None`, in which case it shall evaluate to `None`. A Boolean may be set to None if not initialized to another value but may not be assigned None thereafter.

Enum: a standard or user-defined integral type symbolically representing integers. It supports functions `size()` and `string()`. See 6.10.

Integer: a small structure representing an integer literal or expression that when evaluated, shall reduce to an integer unless it performs an illegal operation or directly or indirectly contains `None`, in which case it shall evaluate to `None`. An Integer may be set to `None` if not initialized to another value but may not be assigned `None` thereafter.

Limits: a small structure used to represent measurement or search limits. With the exception of keyword `None`, which is recognized in a context unambiguously expecting type `Limits` as open limits, literal limits are of general form:

limits ::= **(***LoLim Op ResultPlaceHolder Op HiLim***)**

e.g., (`0s <= .I. < 10ns`),[17] where the enclosing parentheses are an integral part. *LoLim* and *HiLim*, the low and high limits, respectively, are mathemetical expressions. A high limit less than the low limit shall result in undefined behavior. With the exception of keyword `None`, a high limit with different units than the low limit shall be illegal.

Comparison operators, *Op* in the general form, shall be constrained to either <= or <.

ResultPlaceHolder may be either `.I.` or `.O.`, for inside or outside limits respectively. Function `check()` takes a mathematical expression argument, a measurement result whose units shall match the limits'. When function `check()` is executed its argument, presumably a measurement result, replaces *ResultPlaceHolder*.

For *ResultPlaceHolder* `.I.`, the common usage, function `check()` returns `PASS` when the measurement result falls inside specified limits or both limits are set to `None`. Otherwise `check()` returns one of `FAIL_UNITS` (measurement result units mismatch), `FAIL_HILIM`, `FAIL_LOLIM`, or

[17] The Limits syntax is not a Boolean expression hence each limit, lower and upper, is an independent mathematical expression, either of which may or may not be quoted.

`INDETERMINATE` (measurement result is `None`), defined by STIL.4 via enumerated type `CheckResult` (see 35.1). The following is an example showing literal Limits and the Boolean equivalent:[18]

```
Literal Limits:       (0.4V <=         .I.        <= 2.4V)
Boolean equivalent:   0.4V <= result && result <= 2.4V
```

For *ResultPlaceHolder* `.O.`, the rare usage, function `check()` returns `PASS` when the measurement result lies either above or below specified limits. Otherwise `check()` returns one of `FAIL_UNITS`, `FAIL_BOTHLIM`, or `INDETERMINATE`. Valid lower and upper limits, i.e., values other than `None` are required. The following is an example showing literal Limits and the Boolean equivalent:

```
Literal Limits:         (0.4V <=         .O.        <= 2.4V)
Boolean equivalent 1: !(0.4V <= result && result <= 2.4V)
Boolean equivalent 2:   (0.4V >  result || result >  2.4V)
```

Limits shall not be reassigned after instantiation to a value other than `None`.

User-settable constraint `Units` restricts acceptable units, potentially to no units. Function `units()` returns the units as a string which shall be empty if there are no units.

```
 1 FlowVariables {
 2     Limits lims = (0s < .I. < 10ns);
 3 }
 4
 5 TestType ParametricAC {
 6     Inherit StdFunctional;
 7     Parameters {
 8         In Limits lims = None { Units "s"; }
 9         Out Seconds result;
10     }
11     TestExec;    // Assigns "failMode", and "result"
12     PostActions {
13         if (lims.check(result) != PASS)
14             failMode = FAIL_PRM;
15     }
16 }
```

Figure 29—Example: limits function "check"

The following text points out salient syntactical features in the example of Figure 29:

- Line 2: defines a variable called lims which represents literal limits `(0s <= .I. < 10ns)`.
- Line 5: defines a test-type called `ParametricAC`. Parameters are initialized when this type is instantiated. Actions are carried out when the instantiation is executed.
- Lines 7–9: defines parameters to illustrate `Limits` usage.
- Line 11: in this example, `TestExec` represents non-STIL.4 code that performs a parametric timing test, assigns `a FailMode` value to parameter `failMode`, and places the measurement result in parameter `result`.
- Line 13: This statement tests if `TestExec` ran and passed but the limits check failed. Executing `check(result)` in effect replaces `Limits` character `.I.` with `result` and returns

[18] There are differences: the Boolean equivalent may be either single-quoted or unquoted in its entirety and returns `True` or `False` for pass or fail respectively. The return value of Limits function `check` is tested for equality to `CheckResult::PASS` to yield the same Boolean result.

`CheckResult` enumeration `PASS` if `result` falls within limits `lims`. The argument, `result` in this example, shall match `lims` unit constraints.

— Line 14: Alters the contents of `failMode` to indicate a parametric test failure, assuming that is preferable to the contents of `failMode` set by `TestExec`, e.g., `FAIL_SETUP` or `FAIL_FNC`.

```
 1 FlowVariables {
 2        Seconds delay = 3ns;   // Used in subsequent example
 3   Const Limits lims1 = None;
 4   Const Limits lims2 = None { Units "V"; }
 5   Const Limits lims3 = NoneV;
 6   Const Limits lims4 = (0s <= .I. < 10ns);
 7         Limits lims5 = lims4;
 8   Const Limits lims6 = (0s <= .I. < None);
 9         Limits lims7 = (0s <= .I. <= 'delay - 1ns');
10   Const Limits lims8 = lims7;
11         Limits lims9 = None { Units ""; }
12 }
```

Figure 30—Example: FlowVariables block limits definitions

When the argument to `Limits` function `check()` is `None`, it returns `INDETERMINATE`. The following line comments for Figure 30 hold true when the argument value is other than `None`:

— Line 2: variable `delay` is defined for use in a subsequent `Limits` example.
— Line 3: limits are open and unconstrained in terms of units. When limits are set to `None` function `check()` returns `True`.
— Line 4: limits are open and constrained to type `Volts`. Function `check()` returns `True` when limits are set to `None`. It shall be illegal to pass a value to function `check()` that does not match the constraint, units of volts in this example.
— Line 5: is the same as line 4 in effect.
— Line 6: limits are imposed and constrained to type `Seconds`.
— Line 7: copies `lims4` but unlike `lims4`, `lims5` may be reassigned at runtime.
— Line 8: single ended limits are imposed and constrained to type `Seconds`.
— Line 9: limits are imposed and constrained to type `Seconds`. Because `lims7` is a mutable variable, the upper limit shall track `delay`. Were `lims7` defined as a constant, expression `delay - 1` would be evaluated at the instantiation of `lims7`.
— Line 10: copies `lims7`; however, because `lims7` is mutable and `lims8` is constant, expression `delay - 1` is transformed to `3ns - 1ns` or `2ns`, at the tool providers' discretion, for the instantiation of `lims8` (expression remains unchanged for `lims7`).
— Line 11: `lims9` is constrained to be initialized by a number without units, e.g., one of type `Real`.

String: a small structure that behaves like an array of characters which may be assigned a literal value, i.e., zero or more characters enclosed in double quotes or an expression. String expressions may be formed using operator `+` for concatenation. STIL.4 shall impose no limits on string length. An uninitialized `String` shall be set to `None` but may not be assigned `None` thereafter. Accessing a non-existent string element shall return `None`.

VecLocation: a small structure used to pass a STIL.4 vector location. Parameters of this type, when assigned a value other than `None`, override `PatternBurst Start` and/or `Stop` locations for the duration of the test. Attribute `LocType` constrains the location format via qualifiers `None` or `LABEL`. The form for the right-hand side value of an assignment of initialization is as indicated in the "Legal values" column of Table 11 where

vecloc ::= `None` | `EndOfBurst` | `EndOfPattern` | *veclabel*

Keywords `EndOfBurst` and `EndOfPattern` may be used to specify stop locations regardless of the location format. `EndOfPattern` stops at the end of the first pattern in the PatternBurst.[19]

Except for keywords `EndOfBurst` and `EndOfPattern`, when the location type is `LABEL`, the `VecLocation` parameter shall match the STIL.0 `Pattern` label verbatim including quotation marks.[20] The start or stop label shall be unique in the set of patterns referenced in the PatList/PatSet.

```
1     VecLocation vloc = None       { Optional; LocType LABEL; }
```

Figure 31—Example: FlowVariables block VecLocation definitions

The following text points out salient syntactical feature in the example of Figure 31:

— Line 1: constrains legal assignment values to labels. `None` is an optional initialization value for any location type.

The examples in Figure 32 are literal initialization value alternatives that may be passed to a test that has a parameter type `VecLocation` named `vloc` provided the values do not violate parameter definition constraints.

```
1    vloc = EndOfBurst; // Keyword
2    vloc = IDDQ1;      // Unquoted vector label
3    vloc = "Q";        // Quoted vector label
```

Figure 32—Example: VecLocation parameter initializations

The following text points out salient syntactical features in the example of Figure 32:

— Line 1: initializes `vloc` to the last vector in the test's associated `PatternBurst` via keyword `EndOfBurst`.
— Lines 2 and 3: initializes `vloc` to label IDDQ1 and "Q" respectively, i.e., any right-hand side that begins with an alphabetic character or a quote shall be interpreted as a label, not a string.

VecRange: a small structure consisting of two vector locations (*vecloc*) which inclusively indicate a range of vectors. The right-hand side of a `VecRange` initialization or assignment has the following form:

vec_range ::= **(***from_vecloc*, *to_vecloc***)**

where *from_vecloc* and *to_vecloc* use the *vecloc* definition in variable type `VecLocation` above. Pattern location *from_vecloc* shall precede or be equal *to_vecloc*. The syntax shall not permit reference by name to a previously defined `VecLocation` since the name would be indistinguishable from a pattern label.

Window: a small structure used to pass a STIL.4 pattern window. The simplest form is a single pane, i.e., rectangle, whose dimensions are signal count and vector count. A window may be thought of as a `VecRange` combined with a *sigref_expr*. When the signal columns in the pattern are not adjacent, the window may be thought of as composed of multiple horizontal panes. The right-hand side of a `Window` initialization or assignment has the following form:

[19] See Clause 11 for additional information on VecLocation/PatternBurst Start/Stop location interplay.
[20] The label is a reference to a pattern memory location, not a string, i.e., assigning a vector location via a variable of type `String` is illegal.

window ::= **(***from_vecloc*, *to_vecloc*, *sigref_expr***)**

where *from_vecloc* and *to_vecloc* use the *vecloc* definition in variable type `VecLocation` above. Pattern location *from_vecloc* shall precede or be equal *to_vecloc*. The syntax shall not permit reference by name to a previously defined `VecLocation` since the name would be indistinguishable from a pattern label.

Among other things, type `Window` may be used to select a subset of a pattern or patternburst to which strobes are to be applied or from which fail data is collected (the `TestType` specifies how its parameters are used). Figure 33 shows alternate definition examples:

```
1    Window win = None;
2    Window win { Optional; } // Parameter, not FlowVariable
3    Window win = (patlabel1, EndOfPattern, databus);
```

Figure 33—Example: FlowVariable window definitions

Table 10 shows the variable types that represent real values. A type listed in this table may be referenced as meta-type *real_var_type* elsewhere in this document. When a type definition is assigned a literal value, the units shall either match or reduce to the indicated units, e.g., statements `Watts w = 1W` and `Watts w = 1A2R` are both legal.

Real values shall have double precision (binary 64-bit) as defined by IEEE Std 754-2008 or better. In mathematical expressions, keyword None shall be an alias for *not a number* (NaN) and behave as defined by IEEE Std 754-2008. In addition to `type()` and `name()`, all real variable and parameter types support functions `magnitude()` and `units()`.

STIL.0 Table 4 shows unit prefixes and corresponding e-notation recognized by STIL.0 and extensions including STIL.2 and STIL.4. A literal value with units may use either prefix or e-notation. A literal value without units may use e-notation. For example, `0.002V`, `2mV`, `2e-3V`, and `20e-4V` are legal and equivalent. Literal expressions `2m` and `2mm` shall be interpreted as 2 meters and 2 milli-meters respectively.

Type `General` is unique in that it may take on any units, combinatorial or otherwise, including no units. However, once initialized to a value other than `None` or assigned a value (`None` is not a legal assignment value), it shall remain constrained to the units of that value for the remainder of its existence. See Figure 34.

```
General g = None;                 // unconstrained until assigned
General g = None { Units "A2"; } // restricts g to amperes squared
General g = 1;                    // restricts g to no units
General g { Units ""; }           // restricts g to no units
Seconds s = None;                 // restricts s to seconds
Seconds s;                        // same as above
Seconds s = 1s;                   // set value and units
```

Figure 34—Example: real FlowVariable definitions

17.6 FlowVariable attributes

Variable attributes (see Table 12) appear following a definition statement, enclosed in braces. A variable attribute is specified by its name followed by zero or more context specific arguments and terminated by a semicolon. For example:

```
1 FlowVariables {       // Global variables
```

```
2     Volts vdd = 3.3V
3         { Permissions RhsReadWrite; Description "Device power"; }
4 }
```

An attribute, if not explicitly specified, takes on its default value.

Table 12—Variable attributes

Attribute	Argument	Default	Purpose
Description	*string*	Empty string	Declare intended use.
Permissions	**ReadWrite | RhsReadWrite | ReadOnly**	**ReadWrite**	Describe safe editing operations for variables: [a] `ReadWrite` permits edits or deletion, `RhsReadWrite` permits right-hand side value edits only, `ReadOnly` permits neither edits nor deletion.
ReInitAt	ASYNC_EVENT_NAME	START	Describe which asynchronous event reinitializes this variable. All variables shall be initialized on LOAD.
SiteSharePer	**Tester | TestHead | Partition**	Not shared	Variables with this attribute shall be shared across all sites on that unit and implicitly be replicated per specified unit. Variables without this attribute shall implicitly be replicated per site.
Usage	**Pgm | Test**	**Pgm**	Top-level variable access: test program only or memory shared between test program and pattern bursts, respectively.
Units	*string*	unspecified	Constrain to units.

[a] A tool may use Permissions to mark variables on whose presence and/or value a test program depends so that a GUI for the tool that reads STIL.4, may warn a user requesting deletion of a required variable or the changing of a required value. This differs from Const which affects variable behavior at runtime.

STIL.1 and STIL.4 need to control access to global variables across independently executing programs, STIL.1 for communication between `PatternBursts` and patterns, STIL.4 potentially for communication between the test program and pattern-bursts/patterns.

Some of these needs are addressed by standard variables in 35.2. STIL.4 keyword `Usage` options shall be constrained to sharing memory with patterns or not, for type `Integer` only. STIL.4 controls global variable access for variables using the following variacounble attribute syntax:

Usage Pgm | **Test**;

Pgm: the variable is accessible to `TestProgram` only, the STIL.4 default.

Test: the variable is shared between the `TestProgram` and each `PatternBurst` or `Pattern` if all of the following conditions are met:

- Both the STIL.4 and STIL.1 variable shall be of type `Integer`.
- Both the STIL.4 and STIL.1 variable shall use the same identifier.
- Both the STIL.4 and STIL.1 variable shall have attribute `Usage Test.`
- The STIL,4 variable shall not have attribute Const
- The STIL.4 variable and the STIL.1 variable shall both be in scope in their respective context, i.e., the STIL.4 variable shall be either in the unnamed top-level `FlowVariables` block or a named

top-level `FlowVariables` block referenced by `TestProgram`, and the STIL.1 variable shall be in the unnamed `Variables` block.[21]

Shared variables are initialized via STIL.4 code and then potentially altered by STIL.1 code. STIL.1 `InitialValue` syntax shall override STIL.4 initial values of `None` only. STIL.1 and STIL.4 set initial values with different syntax as shown in the `FlowVariables` and `Variables` blocks in Figure 35.

```
 1 FlowVariables {               // STIL.4
 2   Const Integer   i1 = 1;
 3   Const Integer   i2 = 2 { Usage Test; }
 4   Integer         i3     { Usage Test; }
 5   Integer         i4 = 4 { Usage Pgm; }
 6   Integer         i5 = 5 { Usage Test; }             // Shared
 7   Integer         i6 = 6 { Usage Test; }             // Shared
 8   Integer         i7     { Usage Test; }             // Shared
 9   Integer         i8     { Usage Test; }             // Shared
10 }
 1 Variables {                   // STIL.1
 2   IntegerConstant i1 := 10;
 3   IntegerConstant i2 := 20;
 4   Integer         i3 { InitialValue 30; }
 5   Integer         i4 { Usage Test; }
 6   Integer         i5 { Usage Test; }                 // Shared
 7   Integer         i6 { InitialValue 60; Usage Test; } // Shared
 8   Integer         i7 { InitialValue 70; Usage Test; } // Shared
 9   Integer         i8 { Usage Test; }                 // Shared
10 }
```

Figure 35—Example: Variables shared between Pattern and Flow

The following line comments related to Figure 35 refer to both `Variables` and `FlowVariables` blocks:

- Lines 2 and 3: STIL.1 type `IntegerConstant` cannot be shared because in STIL.1, it cannot have attribute `Usage Test`.
- Line 4: variable `i3` is not shared because only the `FlowVariables` block entry associates attribute `Usage Test`. The STIL.1 variable is initialized to zero by default. The STIL.4 variable is set to `None` by default
- Line 5: variable `i4` is not shared because only the `Variables` block entry associates attribute `Usage Test`. The STIL.1 variable is initialized to zero by default.
- Line 6: variable `i5` is shared because both the `FlowVariables` and the `Variables` block entries associate attribute `Usage Test`. Since the variable memory is first set aside in the test program via STIL.4 syntax, it is initialized to 5 and retains that value until altered from within an executing test or pattern.
- Line 7: variable `i6` is shared because both the `FlowVariables` and the `Variables` block entries associate attribute `Usage Test`. Since the variable memory is first set aside in the test program via STIL.4 syntax, it is initialized to 6 and retains that value until altered from within an executing test or pattern.
- Line 8: variable `i7` is shared because both the `FlowVariables` and the `Variables` block entries associate attribute `Usage Test`. Since the variable memory is first set aside in the test program via STIL.4 syntax, it is initialized to `None` until the start of the first pattern execution imposes the STIL.1 initial value of 70.

[21] Named STIL.1 Variables blocks become local to each context they are referenced from, `PatternBurst` or pattern reference block, and are reinitialized each time that context is executed.

— Line 9: variable `i8` is shared because both the `FlowVariables` and the `Variables` block entries associate attribute `Usage Test`. It is not initialized in the `FlowVariables` block, i.e., set to `None`. It is set via the `Variables` block where the absence of an `InitialValue` statement causes default initialization to 0.

17.7 FlowVariable operators and member functions

Table 13—Operator precedence and associativity

Operator	Associativity	Application
Global.	left to right	Scope resolution: global as per TestProgram block
Parent.	left to right	Scope resolution: test or flow that contains current test object
Local.	left to right	Scope resolution: current test object
CurrentExec.	left to right	Scope resolution: current FlowNode's TestExec test object
::	left to right	Scope resolution: Enum definition enumerations and size()
()	left to right	Expression grouping and function call argument delimiter
[]	left to right	Array access and definition
.	left to right	Object member selector
!	right to left	Unary Boolean NOT operator
+	right to left	Unary PLUS operator
-	right to left	Unary MINUS operator
*****	left to right	Binary MULTIPLY operator
/	left to right	Binary DIVIDE operator
%	left to right	Binary MODULO operator (integer division remainder)
+	left to right	Binary ADD operator: addition or string concatenation
-	left to right	Binary SUBTRACT operator
<	left to right	Binary LESS THAN operator
<=	left to right	Binary LESS THAN or EQUAL operator
>	left to right	Binary GREATER THAN operator
>=	left to right	Binary GREATER THAN or EQUAL operator
==	left to right	Binary Boolean EQUAL operator
!=	left to right	Binary Boolean NOT EQUAL operator
&&	left to right	Binary Boolean AND operator
||	left to right	Binary Boolean OR operator
=	right to left	Assignment operator
?:	right to left	Conditional expression operator. Predicate shall be Boolean. Colon operator separates two *int_expr*, *bool_expr*, *string_expr*, *real_expr*, or *enum_expr*. The last two shall have matching units or types respectively.
,	left to right	A function argument and grouped expressions separator [a]
..	left to right	Signal range operator

[a] Exception: `Limits` grouped arguments are separated by relational operator < or <=.

Table 14—FlowVariable member functions

Function	Return type	Purpose	Member function of type(s)
at(*int_expr***)**	`Signal`	Returns the Nth signal in a signal group as specified by *int_expr*.	`SignalGroup`
check(*math_expr***)**	`CheckResult`	Compares the mathematical expression argument to limits and returns an enumeration of type `CheckResult`.	`Limits`
description()	`String`	Returns attribute `Description`.	All variables
hi()	*math_expr*	Returns upper limit.	`Limits`
lo()	*math_expr*	Returns lower limit.	`Limits`
locType()	`LocType`	Returns vector location type as enumerated type `LocType`.	`VecLocation`
magnitude()	`Real`	Returns real value stripped of units. Requires evaluation of mathematical expression.	All real variables (see Table 4)
max()	*math_expr*	Returns mathematical expression contained in `Max`.	`SpecVariable`
meas()	*math_expr*	Returns mathematical expression contained in `Meas`.	`SpecVariable`
min()	*math_expr*	Returns mathematical expression contained in `Min`.	`SpecVariable`
name()	`String`	Returns the variable/parameter instance identifier. For InOut parameters (references), returns the name of the object being referenced, not the local parameter name.	All variables and Parameters
size()	`Integer`	Usually paired with function `at` or operator [] for indexing limits.	`Spec`, `Category`, `SignalGroup`, `Enum`, `String`, BinSpec (when appropriate)
string()	`String`	Returns enumerated type name as a string.	`Enum`
typ()	*math_expr*	Returns mathematical expression contained in `Typ`.	`SpecVariable`
type()	`String`	Returns the variable/parameter type.	All variables
units()	`String`	Returns canonical form such that when the units of two symbols are equal, their unit strings are equal. Each type's return value is shown in Table 4, e.g., for a variable of type Amperes, the function returns "A", for a variable with no units, "", the empty string.	`SpecVariable`, `Limits`, and all real variables (Table 4)

`BinSpec` may represent a `Pass` or `Fail` bin grouping, `a Bin`, or a `BinAxis; f`or related functions see 22.1, 22.3,22.5, and 22.6.

A mutable `SpecVariable`'s `Meas` selector may serve as the target of an assignment. A `SpecVariable`'s `meas()` function, although it returns the `Meas` value, shall not be the legal target of an assignment (see Figure 36).

```
1 specvar.Meas   = 100ps;
2 specvar.Meas   = specvar.typ();
```

Figure 36—Example: SpecVariable field assignment

17.8 FlowVariable array operations

Array operations include the following:

a) Initialization/assignment operator (=)
b) Index operator ([]), index constrained from 0 to array size – 1
c) Mathematical, Boolean, String, and `None` relational operators (`==` `!=`)
d) Mathematical relational operators (`>` `<` `>=` `<=`)
e) Mathematical operators (`+` `-` `*` `/`)
f) Mathematical functions min, max, abs, and pow
g) Boolean operators (`!` `&&` `||`)
h) String operator (+)
i) Array member function size()

For array on array operations, element operates on corresponding element; therefore, left-hand side and right-hand side shall have matching dimensions. For relational operators, e.g., <, all element comparisons shall yield `True` to yield scalar `True`. Functions `min` and `max` also return a scalar. Functions `abs` and `pow` and the remaining operators return an array.

For operations involving scalar and array, the scalar shall operate on each element. For relational operators, e.g., <, all element comparisons shall yield `True` to yield scalar `True`. Functions `min` and `max` also return a scalar. Functions `abs` and `pow` and the remaining operators return an array.

Scalar or array mathematical operations involving `None`, shall evaluate to `None`.

size(): returns a single dimensional array containing one element per dimension, each an integer indicating the size of the corresponding dimension, e.g.:

```
1 FlowVariables {    // Array definitions
2   Real real[4] = [ 4.1, 4.2, 4.3, 4.4 ];
3   Seconds seconds[2][3] = [
4     [ 11ns, 22ns, 33ns ],
5     [ 21ns, 22ns, 23ns ],
6   ];
7 }

1 TestType Example
 2 {
 3   Inherit TestBase;
 4   Parameters {
 5           InOut Const Real    r[4];  // Cannot change contents of r
 6           InOut Const Seconds s[2][3];
 7           InOut       Integer i[][]  { Optional; }
 8           InOut       Amperes a[][2] { Optional; }
 9   }
```

```
10   FlowVariables {
11     Const Boolean idef    = i == None;
12     Const Integer rdims[] = r.size(); // rdims.size=1, rdims[0]=4
13     Const Integer sdims[] = s.size(); // sdims[0]=2, sdims[1]=3
14     Const Integer idims[] = i.size(); // depends on actual dims of I
15           Integer adims[] = a.size(); // depends on actual dims of a
16   }
17 }
```

Figure 37—Example: array size

18. Device to tester interface

This clause covers two alternative methods of describing the device to tester interface, `SignalMap` and `Device` block:

— The `SignalMap` describes the interface between one device and one tester, single or multi-site. It is described in Clause 19. When one or more `SignalMap` blocks are defined, the `TestProgram` block may refer to one of them. The reference, a `SignalMap` statement, specifies which `SignalMap` block shall be used to map the device signals to tester channels.[22]

— The `Device` block describes the interface between one device and one or more testers, single or multi-site. Multiple device blocks may specify multiple test programs to execute in parallel on the same tester, e.g., on different test-heads and/or software partitions. It is described in Clause 20. The `Device` block represents a superset of `SignalMap` block information. It may be employed by ATE as an alternative to `SignalMap`. `Device` block information organization is also suitable for retargeting and ATPRG which can use it to output a `SignalMap` block for each target tester. If a `TestProgram` block does not refer to a `SignalMap` and one or more `Device` blocks are defined, each `Device` block shall choose which test-program to run on a specific tester, test-head, or test-head partition.

[22] A `Device` block, even if it contains an applicable `ChannelMap`, shall not be used when the `TestProgram` block contains the `SignalMap` statement.

Table 15— SignalMap/Device block comparison

Support	SignalMap	Device	Comment
Components		✓	May specify simple hardware components, e.g., capacitors, relays, resistors.
DCSequence per test-head		✓	May specify unique initialization sequence per hardware configuration.
Device families		✓	A single test-program may be used to test a device family, e.g., a set of scalable devices
Multi-chip module (MCM) or multi-chip package (MCP)		✓	Keyword Package may be used to specify the unit containing multiple chips.
Multi-partition		✓	A test-head may be divided into partitions by tester software, each partition potentially testing a different device
Multi-project wafer		✓	A wafer may have two or more chip types to be tested in parallel.
Multi-site	✓	✓	Multiple chip or package sites of the same type may be tested in parallel
Multi-tester		✓	Ability to specify how device is to be tested on more than one platform.
Multi-test-head		✓	Ability to specify how one or more devices are to be tested on more than one test-head on a single tester.

19. SignalMap

19.1 General

A `SignalMap` block maps device signals to one or more tester channels (where tester channels include, but are not limited to, digital channels or power supply channels, for instance) and optionally, to one or more package pins or chip pads. It is possible to map multiple device pins to a single tester channels, and to map a single device pin to multiple tester channels (e.g., power supply or channel ganging).

The `SignalMap` block maps signals to tester channels for one site, or for multiple sites. When mapping for multiple sites, the `SiteMap` statement shall be used; when used, the association between elements of the list of tester channels in *sig_map_stmt*, site identifiers (in the list of sites) and site layout positions is one-to-one, left-to-right.

Only one unnamed `SignalMap` block shall be allowed in STIL. Zero or more named `SignalMap` blocks are allowed. All SIG_MAP_NAMEs shall be unique across all `SignalMap` blocks (STIL.0 6.9: “All domain names for a single type of block shall be unique”). Note, in other words, that domain name reuse for different types of blocks is allowed (i.e., one can use the same domain name for `SignalMap` blocks as is used for a `PatternBurst` block, for instance). See STIL.0 6.9 and 6.16 for details on domain names and domain name conflict resolution.

A named `SignalMap` block is used in a `TestProgram` by including the statement

SignalMap SIG_MAP_NAME**;**

within the `TestProgram` block, as shown below:

```
TestProgram myTestProgram {
    SignalMap mySingleSiteSignalMap;     . . .
}
```

The unnamed `SignalMap` block can be specified in the `TestProgram` block by using the statement `SignalMap Unnamed`. If the `TestProgram` block contains a `SignalMap` statement, the SignalMap shall be used even if a `Device` block with its associated `PinMap` statement also exists.

19.2 SignalMap syntax

SignalMap (SIG_MAP_NAME) {
  (**SiteMap** *site_map_stmt*)
  (*sig_map_stmt*)*****
}+

site_id ::= *integer*
site_id_range ::= *site_id..site_id*
site_list_stmt ::= *site_id* | *site_id_range*(,*site_id* | *site_id_range*)*****
site_map_stmt ::= *site_list_stmt;* | *site_list_stmt* { (**Layout** (SITE_LAYOUT_NAME) (: *site_layout*);) }
sig_map_stmt ::= (SIG_NAME ((P_NAME (, P_NAME)*****)) *chan_stmt* (,*chan_stmt*)***;**
chan_stmt ::= CHAN_ID | CHAN_ID (+CHAN_ID)+;

SignalMap: the start of a block defining device-signal-to-tester-channel mappings (where tester channels include, but are not limited to, digital channels or device power supplies).

SIG_MAP_NAME: optional name of a `SignalMap` block.

SiteMap: an optional statement that defines how many sites are used and the numbers of those sites. If the `SiteMap` statement is omitted, then single-site operation is assumed. The total number of sites specified in the `SiteMap` statement determines the number of sites being used, and the number of tester resources which shall be specified for each signal mapping statement. For instance, with a list of sites such as 1,3,7..10,12,14,16..18,20, a total of 12 sites are specified.

site_id: an integer zero or greater.

site_list_stmt: a comma-separated list of *site_id*s or *site_id* ranges. *site_id* ranges are specified by two ellipsis-separated integers (i.e., *integer1..integer2*), where *integer1* shall be less than *integer2*.

site_map_stmt: a *site_list_stmt*, either semi-colon-terminated, or including an optional site layout statement. See Clause 20 for keyword `Layout` and Figure 47 for details of the `Layout` statement. The number of site grid positions listed in the `Layout` statement shall be the same as the number of sites listed in the *site_list_stmt*, and the association of site number listed in the *site_list_stmt* to the grid position of a site in an MxN grid shall be 1 to 1, left to right.

sig_map_stmt: each signal mapping statement is an individual signal name followed (optionally) by package pin information, and tester channels(s) to which the signal is mapped. The tester channel(s) can be either a single channel statement (*chan_stmt*) (for single-site testing) or a comma-separated list of channel statements (for multi-site testing). If multi-site testing is specified, the number of channel statements in the comma-separated list shall be the same as the number of sites specified in the `SiteMap` statement, and the association of tester resources to sites shall be 1 to 1, left to right.

SIG_NAME: a valid STIL signal name as defined in the unnamed `Signals` blocks (see STIL.0 6.10).

P_NAME: optional parentheses-enclosed physical name or comma separated list of physical names. If used, these names shall have the characteristics of *alnum_id*.

chan_stmt: a single CHAN_ID or a list of CHAN_IDs. A single channel is specified via CHAN_ID (see below). Ganged channels,[23] usually power supplies, may be specified by separating two or more CHAN_IDs with a plus sign.

CHAN_ID: a tester channel identifier as specified by the ATE manufacturer. A channel identifier may take one of the following forms:

- Keyword `None`, which means that no tester resource is connected to the signal.
- Metatype *alnum_id* as defined in 6.5, meaning that double quotes may be used to include otherwise unacceptable characters in the channel name, e.g., `"MCB231,11,o"`. Note that unlike in STIL.0 6.10, which states that there is a distinction between quoted and unquoted signal names, in this case, the quotes are NOT part of the actual channel name.
- An at-sign followed by an unsigned integer may be used to specify that a previously listed channel is used, e.g., mappings

```
sig_a 1f16, 1f16, 1f22, 1f22;
```

and

```
sig_a 1f16, @1, 1f22, @3;
```

are equivalent.

The at-sign notation unequivocably states the intention: field 2 is connected to the same channel as field 1 and field 4 is connected to the same channel as field 3.

[23] A ganged channel specification enumerates channels, each with a separate physical conduit a.k.a. a pogo pin. Channels that are ganged within the tester by software and whose interface is a single conduit shall be specified by a single channel identifier, presumably the master channel.

19.3 SignalMap examples

```
Signals {
  DIR In;
  OE_ In;
  A0 InOut;
  A1 InOut;
  A2 InOut;
  A3 InOut;
  A4 InOut;
  A5 InOut;
  A6 InOut;
  A7 InOut;
  B[0..7] InOut;
  VCC Supply;
  VSS Supply;
}

// Single site – FOR BREVITY, NOT ALL SIGNALS ARE SHOWN
SignalMap mySingleSiteSignalMap {
  DIR ("1") 1;
  OE_ ("19") 10202;
  A0 ("2") A2;
  A1 ("3") "1609.115"; // MUST be quoted; CHAN_ID contains a period (.)

  . . .

  A6 ("8") "8"; // Quoted – but same channel resource ID as if unquoted
  A7 ("9") None;// No connection to tester channel
  B[0] (18) 20;
  B[1] (17) 21;

  . . .
```

```
  B[6] (12) 26;
  B[7] (11) None; // No connection to tester channel
  VCC ("20") DPS1;
  VSS ("10") GND;
}
```

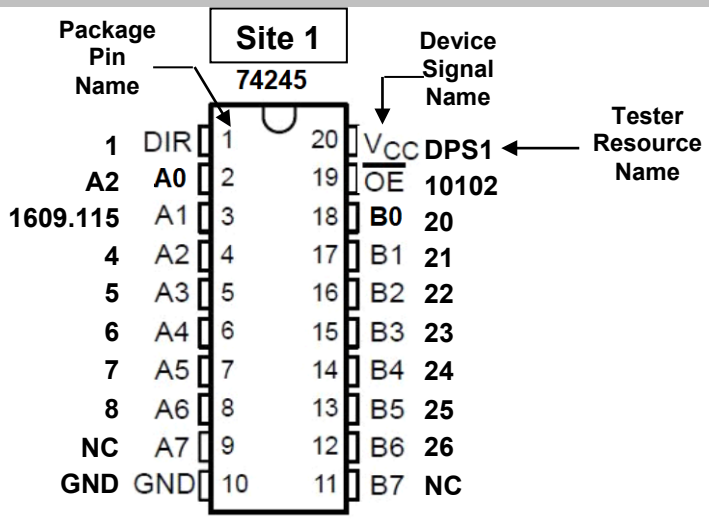

Figure 38—Diagram: single-site SignalMap with Signal names and device pins assigned to tester resources

```
// Multi site - FOR BREVITY, NOT ALL SIGNALS ARE SHOWN
SignalMap myDualSiteSignalMap {
  SiteMap 1,2;
  DIR ("1") 1, 101;
  OE_ ("19") 10202, 20202;
  A0 ("2") A2, B2;
  A1 ("3") "1609.115", "2609.115";

  . . .

  A6 ("8") "8", "108";
  A7 ("9") "9", "109";
  B[0] (18) 20, 120;
  B[1] (17) 21, 121;

  . . .

  B[6] (12) 26, 126;
  B[7] (11) 27, 127;
  VCC ("20") DPS1+DPS3, DPS2+DPS4;
  VSS ("10") GND, GND;
}
```

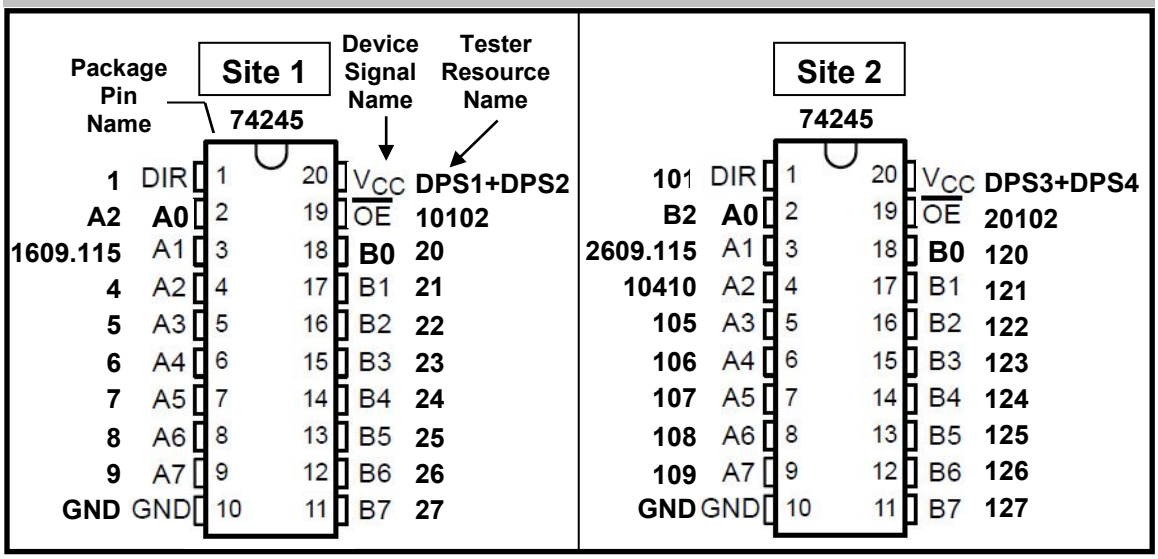

Figure 39—Diagram: multi-site SignalMap with Signal names and device pins assignments to tester resources

```
// Single site, showing multiple device pins (pads) connected to a
// single device signal and tester resource
SignalMap {
  DIR ("1") 1;
  OE_ ("19") 10202;
  A0 ("2") A2;
  A1 ("3") "1609.115"; // MUST be quoted; CHAN_ID contains a period (.)
  . . . // Remainder of signals not shown
  A6 ("8") "8";
  A7 ("9") "9";
  B[0] (18) 20;
  B[1] (17) 21;

  . . . // Remainder of signals not shown

  B[6] (12) 26;
  B[7] (11) 27;
  VCC ("20","22") DPS1; // Two pads, connected to one tester resource
  VSS ("10","21") GND;
}

// Multi site, showing multiple device pins (pads) connected to a
// single device signal and tester resource
SignalMap {
  SiteMap 1,2;
  DIR ("1") 1, 101;
  OE_ ("19") 10202, 20202;
  A0 ("2") A2, B2;
  A1 ("3") "1609.115", "2609.115";

  . . . // Remainder of signals not shown

  A6 ("8") "8", "108";
  A7 ("9") "9", "109";
  B[0] (18) 20, 120;
  B[1] (17) 21, 121;

  . . . // Remainder of signals not shown

  B[6] (12) 26, 126;
  B[7] (11) 27, 127;
  // Two VCC pads, connected to a ganged tester resource on each site
  // (Signal VCC, pins 20 and 22, is connected to DPS1+DPS3 for site 1
  // and DPS2+DPS4 for site 2). Resources for 2nd site are quoted, but
  // the meaning is the same as if they were unquoted.
  VCC ("20","22") DPS1+DPS2, "DPS3"+"DPS4";
  // (Signal VSS, pins 10 and 21, is connected to GND for site 1
  // and GND for site 2).
  VSS ("10","21") GND, GND;
}

// Multi site, showing various forms of the site_map_stmt, with comma-
// separated lists of sites and site ranges in SiteMap statement.

// Site list with lists and ranges
SignalMap {
  // Sites 1,2,4,6,7,8,11,13,14,15 - 10 sites total
```

```
  SiteMap 1,2,4,6..8,11,13..15;
  . . .
}

// Multi site with site layout specified
// Site layout name (QuadDiagLR) is optional.
SignalMap myQuadDiagSiteSignalMap {
  SiteMap 1,2,3,4 {
                  Layout QuadDiagLR: A1, B2, C3, D4;
                  }

  DIR ("1") 1, 101, 201, 301;
  OE_ ("19") 10202, 20202, 30202, 40202;
  A0 ("2") A2, B2, C2, D2;

  . . .  // Remaining signals not show

  VCC ("20") DPS1, DPS2, DPS3, DPS4;
  VSS ("10") GND, GND, GND, GND;
}
```

Mapping of site to site grid layout position to tester channel mapping is one-to-one, left-to-right (mapping for signal DIR shown below)

```
Site 1: grid layout position A1, tester channel 1
Site 2: grid layout position B2, tester channel 101
Site 3: grid layout position C3, tester channel 201
Site 4: grid layout position D4, tester channel 301
```

	1	2	3	4
A	Site 1			
B		Site 2		
C			Site 3	
D				Site 4

Figure 40—Diagram: multi-site SignalMap with diagonal site layout specified, showing assignment of sites to grid positions in 4x4 grid

```
// Multi site with site layout specified
SignalMap myQuadDiagSiteSignalMap {
  SiteMap 1,2,3,4 { Layout QuadSquareCCW: A1, B1, B2, A2; }
  // Site 1 maps to grid position A1, site 2 to grid position B1,
  // site 3 to grid position B2, and site 4 to grid position A2.
  DIR ("1") 1, 101, 201, 301;
  OE_ ("19") 10202, 20202, 30202, 40202;
  A0 ("2") A2, B2, C2, D2;

  . . . // Remainder of signals not shown

  VCC ("20") DPS1, DPS2, DPS3, DPS4;
  VSS ("10") GND, GND, GND, GND;
}
```

	1	2
A	Site 1	Site 4
B	Site 2	Site 3

Figure 41—Diagram: multi-site SignalMap with counterclockwise (CCW) site layout specified, showing assignment of sites to grid positions in 2x2 grid

20. Device (FlowExtended)

20.1 General

A device may consist of a single chip or a package containing one or more chips including chip-to-package connections. This clause describes a device and its connections to one or more testers. The syntax described here may be used by an ATPRG or as an alternative to `SignalMap` when STIL.4 is used as the run-time language on a tester.

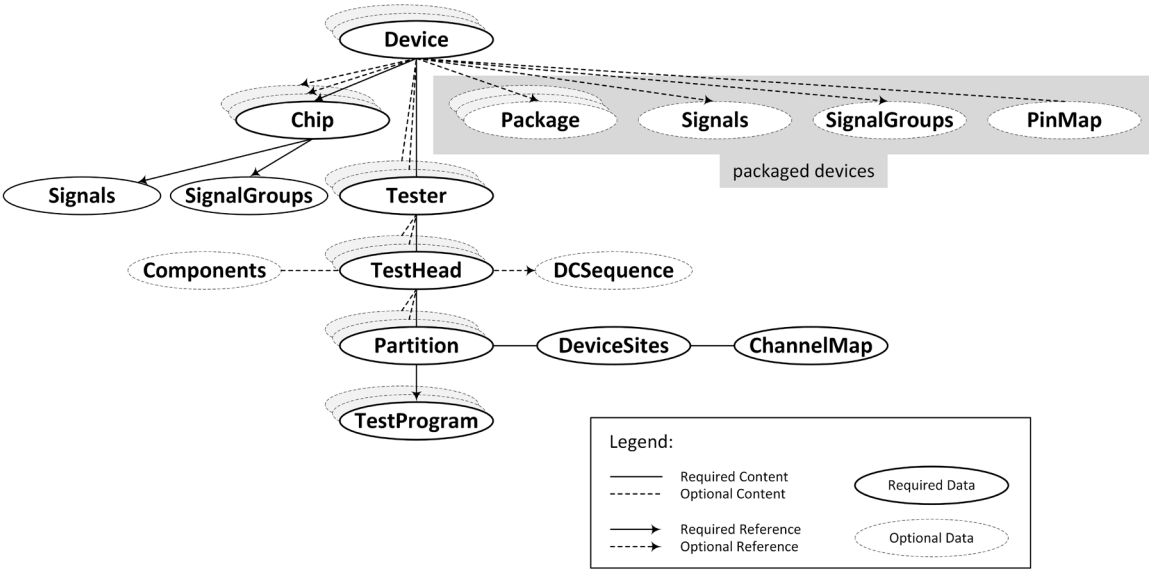

Figure 42—Diagram: Device block overview

A `Chip` block defines per-chip signals and signal groups by reference to top-level named or unnamed `Signal` and `SignalGroup` blocks. The reference to unnamed `Signal` and `SignalGroup` blocks is implicit. The effect of having both named and unnamed blocks is additive. A `Package` block defines per package pins and planes. A `Device` block describes a device in terms of its composition, i.e., one or more chips possibly packaged. Among other things, a `Device` block may specify the following:

— Signal-to-channel mapping
— Signal-to-package-pin or -plane mapping
— The test program used to test the device
— Per tester or test-head DCSequences
— Limited per test-head loadboard components
— Multi-site, MPW, and MCP testing

`Chip`, `Package`, and `Device` blocks are defined at the top level. At least one `Chip` and one `Device` block is required to run a test program. Tester specific information is stored under the `Device` block in a hierarchy that consists of `Tester`, `TestHead`, and `Partition`. Of these, `Tester` and `TestHead` refer to hardware. `Partition` refers to the software partitioning of a `TestHead`. Each `Partition` may run a different test-program (MPW or MCP) controlling a subset of test-head channels. A `Partition` may test one or more devices (multi-site).

```
(Device DEVICE_NAME {
   (Chip CHIP_NAME;)+                          // One chip or 1+ chips in a package or on a MPW
   (Package PACKAGE_NAME;)
   (Signals SIGNALS_NAME;)                     // Usually for open pin signals
   (SignalGroups SIGNAL_GROUPS_NAME;)          // Usually for open pins signal group
   (PinMap { (SIG_NAME <PIN_NAME | PLANE_NAME>(,<PIN_NAME | PLANE_NAME>)*; )+ })
   (Components { (mounted_on { (element_stmt)* })* })*
   (Tester TESTER_NAME {
      (Components { (mounted_on { (element_stmt)* })* })*
      (DCSequence DCSEQ_NAME { … })*
      (TestHead (TESTHEAD_NAME) {
         (Components { (mounted_on { (element_stmt)* })* })*
         (DCSequence DCSEQ_NAME { … })*
         (Partition (PARTITION_NAME) {
            TestProgram TEST_PROGRAM_NAME;
            DeviceSites SITE_NR (, SITE_NR)* {      // Per Tester site numbers
               (Layout (SITE_LAYOUT_NAME) (: site_layout);)
               channelmap_def                       // See 20.5
            }  // End DeviceSites
         })+ // End Partition
      })+ // End TestHead
   })+ // End Tester
})+ // End Device
```

mounted_on ::= < **Loadboard** | **Probecard** | **DUTBoard** >);
A specific *mounted_on* keyword shall only be used once in the `Components` block, e.g., having two `Components` sub-blocks labeled `Loadboard` is illegal.

element_stmt ::=

Capacitor	NAME	{ **Connect** *sigs*; **Value** FARADS; (**Tolerance** (FARADS|PERCENT);) (**Rating** VOLTS;) } |
Diode	NAME	{ **Connect** *sigs*; (**Rating** AMPERES;) (**Vdrop** VOLTS;) (**Vbreak** VOLTS;)} |
Inductor	NAME	{ **Connect** *sigs*; **Value** HENRIES; (**Tolerance** PERCENT;) (**Rating** AMPERES;) } |
Resistor	NAME	{ **Connect** *sigs*; **Value** OHMS; (**Tolerance** PERCENT;) (**Rating** WATTS;) } |
Switch	NAME	{ **Connect** *sigs*; **Type NO|NC,SPST|SPDT|DPST|DPDT**; } |
Wire	NAME	{ **Connect** *sigs*; }

sigs ::= (signal_name|**.C.**)+
PERCENT in this case signifies tolerance, e.g., 5% means ±5%.

Of the names used in the `Device` block BNF syntax description, the following shall use the *alnum_id* syntax described in 6.5: CHAN_ID, CHIP_NAME, DEVICE_NAME, PACKAGE_NAME, PARTITION_NAME, SITE_LAYOUT_NAME, and TESTHEAD_NAME. Previously defined objects adhere to the naming conventions defined in their respective standards hence references to them via DCSEQ_NAME, SIGNALS_NAME, SIGNAL_GROUPS_NAME, and SIG_NAME shall use *name_segment* syntax, whereas TEST_PROGRAM_NAME shall use *simple_identifier* syntax[24]. TESTER_NAME although a STIL.4 construct, shall use *name_segment* syntax for reasons described below under `Tester`.

channelmap_def: this metatype refers to the `ChannelMap` syntax described in 20.5.

[24] Complexity arising from the fact that STIL.4 supports strings (characters surrounded by double quotes) is greatly reduced by avoiding double quoted STIL.0/2/4 identifiers when possible. The addition of STIL.4 naming syntax *alnum_id* reduces the need for enclosing identifiers in double quotes.

Chip: one or more `Chip` statements, each representing an instance of a chip, shall be present.

When followed by keyword `Package`, the device consists of a package containing all previously listed chips, be they the same or different.

When keyword `Package` is absent, the `Chip` statement shall refer to a chip on a wafer. Multi-site testing of a single chip type requires only one `Chip` statement. Sites are specified via keyword `DeviceSites` under `Device/Tester/TestHead/Partition`. Multi-project wafer test shall be specified via multiple `Chip` statements. The first `Chip` statement represents device site 1, the second site 2, etc.

CHIP_NAME: a reference to a previously defined `Chip` block. This block carries with it `Signals` and `SignalGroups` definitions which become visible to any `TestProgram` (including program related timing and levels) specified within the `Device` block. To permit, e.g., two chips in one package with both having VDD tied to a single package plane, identically named signals from two or more chips shall be permitted provided they are electrically and functionally equivalent. Physical signal attribute differences such as pad count, pad number, coordinates, and buffer instance names are not a factor in establishing electrical and functional equivalency.

Components: this keyword introduces an optional block describing simple loadboard, probecard, or DUT-board component connections. This block appears in three places. Via context, one is device specific, one is tester specific, and one is test-head specific. For a particular test-head, these blocks are additive. Each block specifies simple circuits, i.e., each device under test (DUT) signal may have one or more components connected in parallel. The other side of the component connection may be another signal, e.g., a pull-up resistor, or the channel associated with the signal, e.g., a series capacitor. For series connections, syntax element `.C.` is used instead of the signal name to specify the connection to the tester-channel associated with the signal via the `ChannelMap`. More complicated circuits, i.e., multiple components connected in series and parallel, quickly lead to the need for a netlist which is outside the scope STIL.4. Allowing no components would prevent ATPRGs from accounting for common simple circuits like resistors, capacitors, and relays hence STIL.4 offers this middle-ground light-weight syntax.

The number of *sigs* specified for each component should match the number of terminals and are positionally significant as shown in Figure 43, Figure 44, Table 16, and Figure 45.

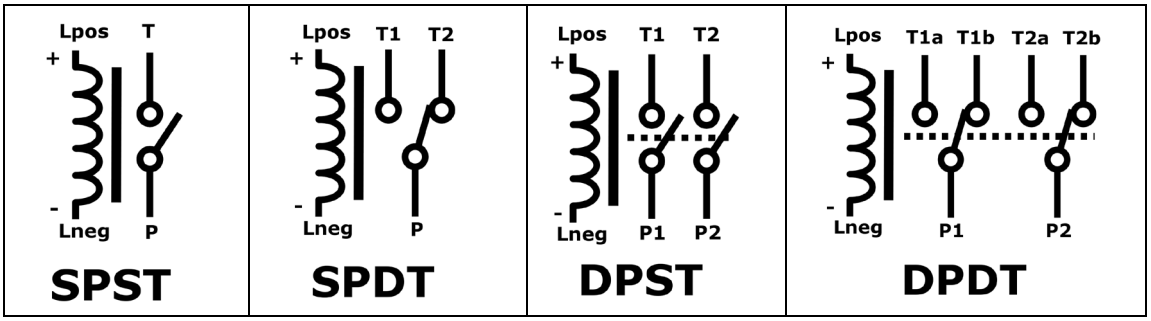

Figure 43—Diagram: relay terminals, normally open positions

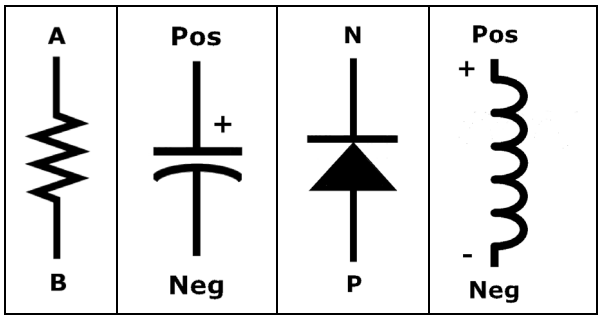

Figure 44—Diagram: component terminals

Table 16—Component-dependent connect statement positional significance

Capacitor	**Connect** Pos Neg;
Diode	**Connect** P N;
Inductor	**Connect** Pos Neg;
Resistor	**Connect** A B;
Switch(SPST)	**Connect** Lpos Lneg P T;
Switch(SPDT)	**Connect** Lpos Lneg P T1 T2;
Switch(DPST)	**Connect** Lpos Lneg P1 T1 P2 T2;
Switch(DPDT)	**Connect** Lpos Lneg P1 T1a T1b P2 T2a T2b;
Wire	**Connect** A B;

```
 1 Components {
 2   Loadboard {
 3     Capacitor C1 { Connect .C. sig_o; Value 8pF;
 4                    Tolerance 0.05pF; Rating 50V; }
 5     Diode     D1 { Connect GND sig_qb;
 6                    Rating 100mA; }
 7     Inductor  L1 { Connect sig_qb sig_o; Value 1mH;
 8                    Tolerance 10%; Rating 100mA }
 9     Resistor  R1 { Connect sig_q .C.; Value 50R;
10                    Tolerance 10%; Rating 0.25W; }
11     Switch    S1 { Connect sig_control GND sig_qb .C. alt_chan;
12                    Type NO,SPDT; }
13     Wire      W1 { Connect sig_q sig_o; }
14   }
15 }
```

Figure 45—Example: loadboard components

The following text points out salient syntactical features in the example of Figure 45:

- Line 1: the `Components` block describes loadboard components and their connections.
- Line 2: this is a keyword indicating the components in the following brace enclosed block are connected on the loadboard.
- Line 3: capacitor `C1`'s contact point `Pos` is connected to the tester-channel(s) associated with device signal `sig_o`. Contact point `Neg` is connected to device signal `sig_o`. A channel per site is associated with a signal in the `ChannelMap` block.
- Line 5: diode `D1`'s contact point `P` is connected to signal `GND`, presumably ground, and contact point `N` is connected to device signal `sig_qb`.

— Line 7: the inductor connects two signals on the device.
— Line 9: resistor `R1`'s contact point `A` connects to device signal `sig_q` and contact point `B` connects to the channel(s) associated with device signal `sig_q`.
— Line 11: loadboard relays are controlled via the `DCSequence` block `Switch` statement explained in Clause 14. For control on a per test basis, see `DCSequence` in 35.6.
— Line 13: a wire is treated as an easily distinguishable 0 Ohm resistor. In this case resistor `W1`'s contact point `A` connects to device signal `sig_q` and contact point `B` connects to the channel(s) associated with device signal `sig_o`.

DCSequence: this keyword introduces a power up/down sequence definition as specified in STIL.2. A `DCSequence` may be defined at the top (global), `Tester`, and `TestHead` levels. Of these, the one closest to the local scope shall apply.

DCSEQ_NAME: represents one of standard names `InitialSetup`, `EndOfProgram`, `PowerRaise`, `PowerLower`,[25] or user-defined `User` USER_DEFINED_NAME. All standard named DC-sequences shall be applied via the execution sequence emanating from entry-point `On START`. `InitialSetup` shall be applied before the first `DCLevels`. `EndOfProgram` shall be applied after entry-point `On START` relinquishes control. `PowerRaise`/`PowerLower` shall be applied if any power-supply levels rise/fall in absolute value of programmed voltage. `PowerRaise`/`PowerLower` shall be applied before the DC levels that would cause that rise/fall are set. If some power supply levels rise while others fall, `PowerRaise`/`PowerLower` shall not be applied. Application of a user-defined sequence shall be specified on a per test basis via its `DCSequence` parameter.

Device: the device name represents one or more chips or a combination chip(s), package, and wire bonding. Multiple device blocks may be specified for the purpose of parallel testing, e.g., on separate test-heads. No two device blocks may have the same name. At the user's discretion the device name may be the same as the chip name, presumably for wafer test. The presence of more than one chip in a device specification indicates a multi-project/product wafer (MPW), multi-chip package (MCP), or multi-chip module (MCM).[26]

DeviceSites: this keyword refers to one or more device sites by number. At wafer test, device sites are chip sites. For multi-project wafers, different chips may occupy individual sites. At package test, device sites are package sites. Although all packages are identical, each package may contain one or more potentially different chips.

SITE_NR: an integer zero or greater. Each site number listed is tested via test program TEST_PROGRAM_NAME. Each device site number shall be unique within the `Tester` block. See `ChannelMap` for signal-to-channel mapping statements.

Layout: this keyword introduces physical, single-layer, tiled, multi-site layout information.

SITE_LAYOUT_NAME: an arbitrary name, normally optional, shall be required to distinguish between two or more `DeviceSites` statements that have the same site count and layout. Examples: horizontal, vertical, diagonal, or probecard or loadboard name.

[25] Even though `DCSequence` statements are optional, it is good practice to define at least `InitialSetup` and `EndOfProgram`. Without these, target tester default behavior shall be invoked. `InitialSetup`/`EndOfProgram` definitions should be paired. `PowerRaise`/`PowerLower` definitions should be paired. Top-level `DCSequence` definitions are discouraged because they apply to every tester that does not have a local definition regardless of its architecture or capabilities.
[26] When multiple cores are wrapped in a hardware structure such as defined by IEEE Std 1500, they should collectively be defined as a single chip. When multiple cores are not wrapped in a hardware structure, they may be defined as individual chips, one per core type.

site_layout: a comma-separated list of site locations each specified using the pin grid array (PGA) pin naming convention (top view). For example, the statement in Figure 46 maps sites 1 and 3 to A1 and B2, i.e., A1 in 1st position maps to site 1, B2 in 2nd position maps to site 3, etc.

```
DeviceSites 1, 3 {
    Layout DiagonalLR: A1, B2;
    ...
}
```

Figure 46—Example: device site layout

The syntax of Figure 46 describes the layout depicted in Figure 47.

	1	2
A	Site 1	
B		Site 3

Figure 47—Diagram: device site layout

Partition: this keyword is used to enumerate software partitions on a test-head. Each partition executes a test-program that controls the channels specified under the `DeviceSites` block active at runtime.

Package: this keyword specifies a package[27] containing all chips previously listed in the `Device` block. The brace enclosed block contains `PinMap` which offers syntax for specifying `Signal`-to-package-pin connections, part of the chip-to-tester connectivity. A semicolon terminated package specification is incomplete however a tool may import these connections from a non-STIL.4 source.

PACKAGE_NAME: a reference to a previously defined `Package` block.

PinMap: this keyword introduces one or more chip-signal-to-package-pin or -package-plane mappings using syntax shown under meta-type *pkg_attributes*. This map facilitates including the package in an ATPRG's `Device` chip-to-tester connectivity.

SIG_NAME: the signal name shall be drawn from the unnamed STIL.0 `Signals` block, `Signals` block(s) referenced by the chip(s) named in the `Device` block, or `Signals` block referenced in the `Device/Package` block. No one of these `Signals` blocks is required but at least one shall be present to have a pool of signal names to draw from.

PIN_NAME: a reference to a package pin name as defined in a top-level `Package` block. Only pin names from the package referenced in the `Device` block shall be used. Keyword `None` may be used to specify a chip signal that is not connected to any package pin.

Package composition and individual mapping statements correlate like this:

— When the package is composed of one chip, a SIG_NAME maps to a PIN_NAME or PLANE_NAME.
— When the package is composed of multiple chips of the same kind, one SIG_NAME maps to two or more PIN_NAME or PLANE_NAME separated by commas. The first PIN_NAME or PLANE_NAME

[27] The package referred to by name may have been defined by the user earlier in the input stream or by an ATPRG library's package description library.

corresponds to the first `Chip` statement, the second PIN_NAME or PLANE_NAME corresponds to the to the second `Chip` statement, etc.

— Exception: a SIG_NAME of signal subtype `Open`[28] always maps to one PIN_NAME or PLANE_NAME.

Signals: this optional keyword introduces a reference to a named `Signals` block usually used to define signals representing package pins not connected to the chip. These signals may be used to test for electrical isolation on unused pins. An ATPRG may deduce this information given the chip's signals and the package `PinMap`. All identifiers within the `SignalGroups` and `Signals` blocks shall share the signal namespace within the `Device` block where they are used. That namespace shall be visible to the `TestProgram` specified lower in the `Device` block hierarchy. Refer to 8.2 for signal details.

SignalGroups: this optional keyword introduces a reference to a named `SignalGroups` block usually used to define signal groups representing package pins not connected to the chip.[29] These signal groups may be used to test for electrical isolation on unused pins. An ATPRG may deduce this information given the chip's signals and the package `PinMap`. All identifiers within the `SignalGroups` and `Signals` blocks brought together here shall share the `Device` block's signal namespace. That namespace shall be visible to the TestProgram specified lower in the `Device` block hierarchy. It is recommended that signal attributes be defined only in the corresponding signals block. Refer to 8.2 for signal details.

Tester: this keyword is used to introduce information which is generally repeated on a per target tester basis to describe, e.g., tester-specific loadboard components, which test-head to use and how it is partitioned.

When more than one configuration of an ATE manufacturer's tester model is available, TESTER_NAME represents one of those configurations. TESTER_NAME shall be unique among tester names within the `Device` block. This allows an ATPRG to determine the resources and constraints behind each tester-channel, presumably from its own database, possibly via STIL.3.[30]

To provide a disciplined method for mapping TESTER_NAME to STIL.3 test resource constraints (`TRC`) and maximum portability between ATPRGs, the following convention is suggested for TESTER_NAME:

"VENDOR_NAME, MODEL_NAME, CONFIGURATION_NAME"

Since TESTER_NAME uses *name_segment* syntax, the quoted id containing comma field separators is legal. The VENDOR_NAME allows an ATPRG targeting that vendor's tester to ignore all other vendor's tester descriptions in its database. The MODEL_NAME allows an ATPRG to target a specific language and implies some generic limitations. An ATPRG may be able to generate a test program on the basis of VENDOR_NAME and MODEL_NAME alone, albeit with limited error checking. The CONFIGURATION_NAME may specify the configuration of a specific tester or a group of testers of which this configuration is emblematic. The CONFIGURATION_NAME is the least portable of the three however it alone is sufficient for test program generation with maximal error checking. Although not recommended, TESTER_NAME containing only the CONFIGURATION_NAME would be two commas followed by the name like this:

",,CONFIGURATION_NAME"

If a model name has major and minor components, separate them with a colon like this:

"VENDOR_NAME**,** MAJOR_MODEL_NAME **:** MINOR_MODEL_NAME**,** CONFIGURATION_NAME"

[28] The signal name in this case is an alias for the package pin that is not bonded to a pad. That signal name may then be used to test the pin to be sure it is isolated.

[29] The norm is to define a single signal-group representing unconnected pins. A signal-group definition in this context may also be used for scalable devices, e.g., a group named DATABUS may contain 32 signals for one device and 64 for another. Both devices may then use the same levels and timing associated with DATABUS.

[30] STIL.4 does not include tester resource and constraint description syntax.

TestHead: this keyword introduces one or more test-head usage descriptions, i.e., the block contains one or more software partitions, each of which controls a set of tester-channels located on that test-head. TESTHEAD_NAME shall be unique among test-head names within the `Tester` block, i.e., when testing the same type of device on multiple test-heads. TESTHEAD_NAME shall be unique among different `Device` blocks when it applies to different devices tested in parallel on different test-heads on the same tester.

TestProgram: this keyword introduces forward reference TEST_PROGRAM_NAME which specifies the test program to use in this context.

20.2 STIL.2: DC levels

STIL.4 supports per tester/testhead specific definitions for `DCSequence InitialSetup`, `PowerRaise`, `PowerLower`, `EndOfProgram`, and `User` in addition to top-level namespace definitions specified in STIL.2 Clause 12. For a `DCSequence` defined under the `Device block`, STIL.4 ignores the optional `SignalGroups` statement. This is because the `Device block` already has the relevant signals and signal groups imported into its environment via its `Chip` and `Package` statements. The use of top-level `DCSequence` definitions is discouraged because different testers likely require different sequences.

20.3 Chip

(**Chip** CHIP_NAME; | **Chip** CHIP_NAME { (*chip_attributes*) })+

Chip: this keyword introduces a required block defining a chip type/design.

CHIP_NAME: this name shall be used to distinguish one chip type/design from another.

chip_attributes ::=
(**Signals** SIGNALS_NAME;)
(**SignalGroups** SIGNAL_GROUPS_NAME;)

The signals associated with `Chip` shall be the sum of the signals defined in the unnamed `Signals` block and the signals defined in the named `Signals` block, which may optionally be referenced. Similar logic applies to `SignalGroups`.

SignalGroups: this optional keyword introduces a reference to a previously defined `SignalGroups` block named SIGNAL_GROUPS_NAME. All identifiers within the `SignalGroups` and `Signals` blocks shall share the signal namespace within the `Device` block where they are used including the unnamed `SignalGroups` and `Signals` blocks. That namespace shall be visible to the `TestProgram` specified lower in the `Device` block hierarchy. It is recommended that signal attributes be defined only in the signals block. Refer to 8.2 for signal details.

Signals: this optional keyword introduces a reference to a previously defined `Signals` block named SIGNALS_NAME. All identifiers within the `SignalGroups` and `Signals` blocks shall share the signal namespace within the `Device` block where they are used including the unnamed `SignalGroups` and `Signals` blocks. That namespace shall be visible to the `TestProgram` specified lower in the `Device` block hierarchy. Refer to 8.2 for signal details.

For example, the following statement associates the sum of definitions from the unnamed `Signals` block, the unnamed `SignalGroups` block, and the `SignalGroups` block named `B2809` with chip `B2809`:

```
Chip B2809 { SignalGroups B2809; }
```

20.4 Package

(**Package** PACKAGE_NAME { *pkg_attributes* })*

Package: this keyword introduces an optional block defining a package or MCM type.

PACKAGE_NAME: this name shall be used to distinguish one package type from another, e.g., `196_BGA`. Specific identifier rules are defined by metatype *stil4_ alnumid* (6.5)

pkg_attributes ::= **PinList** (PIN_COUNT) { *pinlist_expr* }
(**Plane** PLANE_NAME { *pinlist_expr* })*

*pkg_attributes***:** enumerates pin and plane names and provides for signal-to-pin mappings. All pin and plane names shall be unique for the package, i.e., they exist in the same namespace. These optional attributes help an ATPRG keep track of connections and generate signals for unused package pins should the need arise to test them for isolation. The user may choose to define unused package pin signals within the `Device` block.

pinlist_expr ::= PIN_NAME | *pinrange_expr* (,PIN_NAME | ,*pinrange_expr*)*
PIN_NAME ::= *pos_int* | *upcase_letter*+ *pos_int*
pinrange_expr ::= *number_range* |
upcase_letter+ [*number_range*] |
[*letter_range*] [*number_range*] |
[*letter_range*] *pos_int*
letter_range ::= *upcase_letter*+**..***upcase_letter*+
number_range ::= *pos_int***..***pos_int*

PinList: optional keyword names all pins associated with the package. The word *pins* is used figuratively to denote contact points, be they actual pins, balls, or something else. Shorthand notation may be used for listing pin names, e.g., `A[1..2]` yields PGA pin names `A1` and `A2`; `[A..C][1..3]` yields PGA pin names `A1`, `A2`, `A3`, `B1`, `B2`, `B3`, `C1`, `C2`, and `C3`. Bracket enclosed ranges shall be ordered alphabetically or numerically from low to high. Range `[Z..AB]1` generates `Z1`, `AA1`, `AB1`, etc. Pins that are enumerated in `PinList` but not associated with a signal are deemed to be unconnected unless mentioned under `Plane` in which case they are connected only to each other. Signal-to-pin association shall be specified in block `Device/Package/PinMap`.

PIN_COUNT: an optional unsigned integer indicating the total number of package pins. This integer may be used as a cross-check for the number of individual pins specified via *pinlist_expr*.

pinlist_expr: one or more pins. The pins are either named individually or generated by a *pinrange_expr*, or a combination of the two.

PIN_NAME: either an integer greater than zero, or upper-case letter(s) followed by an unsigned integer.

pinrange_expr: a range of two or more pin names, each of which shall conform to the PIN_NAME definition.

letter_range: range shall be from low to high. The first range element shall be lexically less than the second.

number_range: range shall be from low to high. The first range element shall be numerically less than the second.

Plane: lists pins, a subset of `PinList`, tied together to form a single electrical node usually for power or ground. The shorthand notation used for `PinList` may be used to enumerate pins in the plane.

PLANE_NAME: a meta-type *simple_identifier* conforming identifier, unique within the `Package` block, e.g., `A`, `B`, `PWR1`, or `GND`.

pinlist_expr: the format is the same as *pinlist_expr* as defined for `PinList` but only pin names from the package `PinList` shall be used. No pin shall appear in more than one plane.

20.5 Channel map

This structure maps signal names to tester-channel names for a particular tester, test-head, and test-head partition.

channelmap_def ::=
ChannelMap {
(SIG_NAME *chan_stmt* (,*chan_stmt*)*; |
SIG_NAME *chan_stmt* (,*chan_stmt*)* { (*chan_config*)* } |
SIG_NAME [*chan_stmt* (,*chan_stmt*)*] ([*chan_stmt* (,*chan_stmt*)*])*; |
SIG_NAME [*chan_stmt* (,*chan_stmt*)*] ([*chan_stmt* (,*chan_stmt*)*])* { (*chan_config*)* })*
}

chan_stmt ::= CHAN_ID | CHAN_ID (+CHAN_ID)+ | CHAN_ID[*num_list*];

num_list ::= < *pos_int* | *pos_int*..*pos_int* > (,< *pos_int* | *pos_int*..*pos_int* >)*

chan_config ::= **ChanDirection** < **In** | **Out** | **InOut** >;

*chan_config***:** distinguishes signal attributes from channel attributes. See `ChanDirection` below.

ChannelMap: this keyword introduces a block defining signal-to-tester-channel mappings. One map per `DeviceSites` block may be defined.

Each mapping is a signal name followed by comma separated channels where the number of channels shall correspond to the number of signals to be mapped. For example, a simple device consisting of one chip in one package requires one channel per signal. Dual site testing usually requires two channels per signal. The first channel corresponds to the first SITE_NR of the `DeviceSites` statement, the second to the second, etc. The mapping may be terminated by a semi-colon or a brace-enclosed block containing a channel configuration.

For wafer test, the `Device` block has one or more Chip specifications and no `Package` specification. If there is only one `Chip` statement in the `Device` block, then instances of that chip are automatically replicated to accommodate specified site numbers. If there is more than one `Chip` statement in the `Device` block, then the first `Device` block `Chip` statement represents site 1, the second site 2, etc. When mapping a signal, `ChannelMap` associates a channel name with each site number by position. For example, given statement `DeviceSites 1,3` the first CHAN_ID applies to site 1, the second to site 3.

For package test, the `Device` block has one or more `Chip` specifications and one `Package` specification. If there is only one `Chip` statement in the `Device` block, then the package contains that one chip. If there is more than one `Chip` statement in the `Device` block, then the packaged device is composed of the chips listed. The chips listed may all be of the same type or not. In this context, statement `DeviceSites 1,3` refers to package sites. To better visualize grouping, an optional alternative mapping syntax enclosing comma separated channel names in brackets is offered. Each set of brackets encloses channels for a single package site. For statement `DeviceSites 1,3` the first set of brackets shall contain channels for package site one, the second set, channels for package site three. The `Device` block specified package and chips are automatically replicated to match the number of sites.

SIG_NAME: the signal name shall be drawn from the unnamed STIL.0 `Signals` block, the `Signals` block(s) referenced by the chip(s) named in the `Device` block, or the `Signals` block referenced in the `Device` block. No one of these `Signals` blocks is required but at least one shall be present to have a pool of signal names to draw from.

chan_stmt: this statement specifies one or more tester-channels. A single channel is specified via CHAN_ID. Ganged channels,[31] usually power supplies, may be specified by separating two or more CHAN_IDs with a plus sign. Alternatively, ganged channels may be specified by a bracketed expression suffix that expands into multiple identifiers, e.g., `PWR[1,3..5]` is equivalent to `PWR1+PWR3+PWR4+PWR5`. Bracketed numeric values shown in metatype *num_list* as *pos_int*, shall be greater than 0 listed in ascending order.[32]

CHAN_ID: a tester-channel identifier as specified by the ATE manufacturer.[33] A channel identifier may take one of the following forms:

- Keyword `None` means that no tester-channel is connected to the signal**.**
- A *alnum_id* as defined in 6.5 means that double quotes may be used to include otherwise unacceptable characters in the channel name, e.g., `"MCB231,11,o"`. Unlike in STIL.0, the quotes are not part of the actual channel name.
- An at-sign followed by an unsigned integer may be used to specify that a previously listed channel is used, e.g., mappings `sig_a 1f16, 1f16, 1f22, 1f22;` and `sig_a 1f16, @1, 1f22, @3;` are equivalent. The at-sign notation unequivocably states the intention: field 2 is connected to the same channel as field 1 and field 4 is connected to the same channel as field 3. The preceding field numbers are equivalent to site numbers if the channel map is enclosed by block `DeviceSites 1,2,3,4`, since each field corresponds to its respective enumerated site.

ChanDirection: this per-signal attribute should be used only when tester-channel directionality (`In`, `Out`, `InOut`) is different from the device signal directionality.[34] Unnecessary use may introduce unintentional errors. Even though this attribute specifies channel direction, its parameter is from a device point of view, i.e., `In` implies that the channel may drive, `Out` implies that channel may compare. This is in keeping with how most testers label the direction of the associated SIG_NAME.

[31] A ganged channel specification enumerates channels, each with a separate physical conduit a.k.a. a pogo pin. Channels that are ganged within the tester by software and whose interface is a single conduit shall be specified by a single channel identifier, presumably the master channel.

[32] For simplicity's sake, metatype *num_list* BNF permits a single member list, e.g., `PWR[1];` however, ganging a single channel should be flagged as an error.

[33] Each tester may have a unique naming scheme. `CHAN_ID` names the conduit, the physical manifestation of which is the pogo pin. The distinction is made because in some cases the tester resource behind the channel may be switched at runtime.

[34] This attribute may be used to constrain a channel to operate in `Out` (comparator) mode for an `InOut` device signal. Alternatively, some testers require that a channel be configured as `InOut` for a particular test-type whereas signal directionality is `In` or `Out`. The separation facilitates retention of device signal directionality information.

20.6 Multi-site/MPW testing

By default all mutable (non-const) objects are implicitly replicated per site. For example, for a tester with one testhead divided into two partitions, both partitions running the same test program, one with 2 sites, the other with 1, each mutable object has 3 instances in effect. The set of mutable objects includes variables, spec-variables, tests, and soft and hard bins.

Variable attribute `SiteSharePer` may be used to define variables that are shared across sites on a per-tester, test-head, or partition basis.

For MPWs, a `Device` block with multiple chips and no package may be defined.

20.7 Device block examples

The `Signals`, `SignalGroups`, `Chip`, and `Package` blocks in Figure 48 are defined for reference by subsequent `Device` block examples:

```
Signals {
  IN0       In;
  IN1       In;
  IO[0..1]  InOut;
  OUT       Out;
  VDD       Supply;
  VSS       Ground;
}
SignalGroups {
  IOS = 'IO[0] + IO[1]';
  ALLSIGS = 'IN0 + IN1 + IO[0] + IO[1] + OUT';
}
Chip B35 {}  // Uses unnamed Signals and SignalGroups blocks
// ------------------------------------------------------------------
Package 16DIP {
  PinList { 1..16 }    // Pin names 1 through 16
  Plane A {  5,  8; }  // Plane A connects pins  5 and  8
  Plane B { 12, 16; }  // Plane B connects pins 12 and 16
}
Signals TWO_B35_IN_16DIP {
  open3  InOut+Open;   // No internal connections on package pin  3
  open4  InOut+Open;   // No internal connections on package pin  4
  open11 InOut+Open;   // No internal connections on package pin 11
  open13 InOut+Open;   // No internal connections on package pin 13
}
SignalGroups TWO_B35_IN_16DIP {
  opens = 'open3 + open4 + open11 + open13';
}
// ------------------------------------------------------------------
Package 6DIP {
  PinList { 1..6 }     // Pin names 1 through 6
}
SignalGroups B35_IN_6DIP {
  opens = '';
}
```

Figure 48—Example: Signals, SignalGroups, chip, and package definitions

The following text points out salient syntactical features in the example of Figure 48:

- Lines 1–8: defines signals for every chip due to the fact that the `Signals` block is unnamed. The unnamed `Signals` block can be used to define common denominator signals for a chip family.
- Lines 9–11: defines signal groups for every chip due to the fact that the `SignalGroups` block is unnamed. The unnamed `SignalGroups` block can be used to define common denominator signal groups for a chip family.
- Line 12: defines fictional chip `B35`. Chip signals and signal groups come from the unnamed `Signals` and `SignalGroups` blocks. Were there a reference to a named `Signals` block within the associated `Chip` block braces, chip signals would be the sum of signals in the unnamed and named `Signals` blocks. The summing would apply also were there a reference to a named `SignalGroups` block.
- Lines 14–18: block defines fictional package `16DIP`. Keyword `Package` can also be used to define a MCM.
- Line 15: enumerates package pin names 1 through 16 via an expandable expression.
- Lines 16–17: defines package planes `A` and `B` where plane `A` is connected to pins `5` and `8` and plane `B` is connected to pins `12` and `16`.
- Lines 19–24: arbitrarily named signals block for subsequent reference.
- Lines 25–27: arbitrarily named signal groups block for subsequent reference.
- Lines 29–31: block defines fictional package `6DIP`.
- Line 33: there are no internally unconnected package pins. Timing and levels associated with empty signal group `opens` are not applied; therefore. the same timing and levels can be used for chip `B35` at wafer or enclosed in different packages.

Figure 49 shows a diagram of a single-side wafer test.

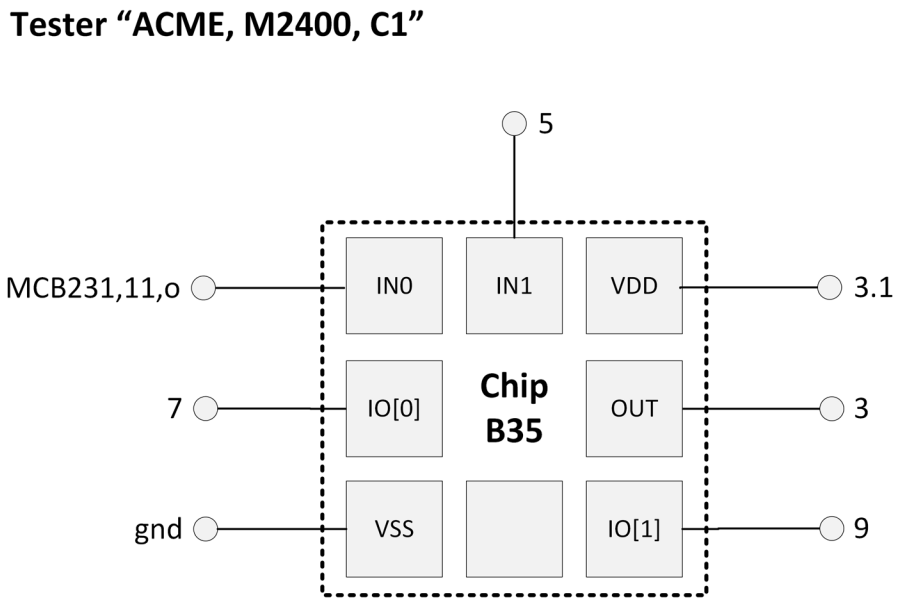

Figure 49—Diagram: single-site wafer test

The `Device` block example in Figure 50 is for single-site testing of a chip.

```
 1 Device B35_WAFER {
 2   Chip B35;
 3   Tester "Verigy, 93000, standard_digital" {
 4     DCSequence InitialSetup { // Beginning of program
 5       '0s'    VDD { Apply '0V'; }
 6       '1ms'   VDD { Connect Supply; }
 7       '100us' VDD { Apply; }
 8       '1ms'   ALLSIGS { Connect Load; Connect Driver Comparator; }
 9     }
10     DCSequence EndOfProgram { // End of program
11       '0s'    'VDD+ALLSIGS' { Ramp '5ms' '0V'; }
12       '6ms'   ALLSIGS { Disconnect Driver Comparator; }
13       '1ms'   VDD { Disconnect Supply; }
14     }
15     TestHead {
16       Partition {
17         TestProgram B35_WAFER;
18         DeviceSites 1 {
19           ChannelMap {
20             IN0    10101;
21             IN1    10102;
22             OUT    10103;
23             IO[0]  10104;
24             IO[1]  10105;
25             VDD    10106;
26             VSS    gnd;
27           }  // End ChannelMap
28         }  // End DeviceSites
29       }  // End Partition
30     }  // End TestHead
31   }  // End Tester
32 }  // End Device
```

Figure 50—Example: Device block for single-site wafer test

The following line comments refer to Figure 50:

- Line 2: reference to a previously defined chip.
- Lines 4–5: these are two `DCSequence` blocks, each followed by a standard id and brace enclosed block. The id and ellipses represent code specified by STIL.2. Both sequences apply to the following `TestHead` statement since overrides, sequences of the same name, are not defined in the `TestHead` block.
- Line 8: keyword `TestProgram` is followed by a forward reference to the test program that is to be executed by this partition. Program `TWO_B35_IN_16DIP` is omitted from this example in order to focus on the device/tester interface.
- Lines 10–18: keyword `ChannelMap` is followed by a brace enclosed block of signal-to-tester-channel mappings. All signals shall be accounted for, i.e., all unnamed `Signals` block signals and any signals from named `Signals` blocks referenced in the `Chip` and `Device/Package` blocks. Channel names used in this example represent different tester naming conventions to point out syntax usage.

Figure 51 shows a diagram of a single-side package test.

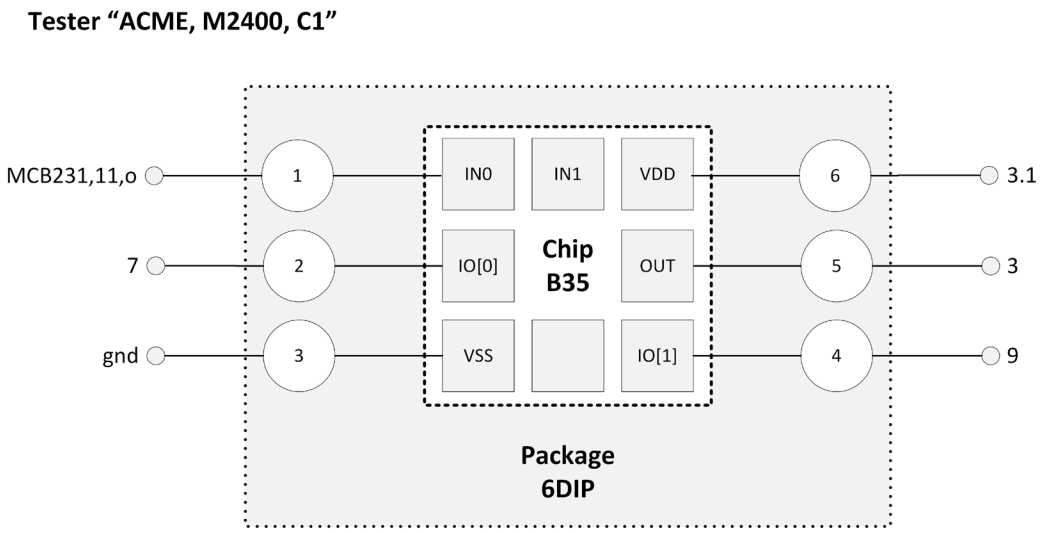

Figure 51—Diagram: single-site package test

The `Device` block example in Figure 52 is for single-site testing of a chip in a package:

```
Device B35_IN_6DIP {
  Chip B35;
  Package 6DIP;
  SignalGroups B35_IN_6DIP;
  PinMap {              // Signal-to-package-pin map
    IN0      1;
    IO[0]    2;
    IO[1]    4;
    OUT      5;
    VDD      6;
    VSS      3;
  }
  Tester "Verigy, 93000, standard_digital" {
    DCSequence InitialSetup { // Beginning of program
      '0s'    VDD { Apply '0V'; }
      '1ms'   VDD { Connect Supply; }
      '100us' VDD { Apply; }
      '1ms'   ALLSIGS { Connect Load; Connect Driver Comparator; }
    }
    DCSequence EndOfProgram { // End of program
      '0s'    'VDD+ALLSIGS' { Ramp '5ms' '0V'; }
      '6ms'   ALLSIGS { Disconnect Driver Comparator; }
      '1ms'   VDD { Disconnect Supply; }
    }
```

```
25     TestHead {
26       Partition {
27         TestProgram B35_IN_6DIP;
28         DeviceSites 1 {  // Device (package) site number
29           ChannelMap {
30             IN0    10101, @1, @1, @1{ ChanDirection InOut; }
31             IN1    None, None, None, None;
32             OUT    10103;
33             IO[0]  10105;
34             IO[1]  10107;
35             VDD    11,13;
36             VSS    gnd;
37           }  // End ChannelMap
38         }  // End DeviceSites
39       }  // End Partition
40     }  // End TestHead
41   }  // End Tester
42 }  // End Device
```

Figure 52—Example: Device block for single-site package test

The following line comments refer to Figure 52:

- Line 2: reference to a previously defined chip.
- Line 3: reference to a previously defined package.
- Lines 4–12: package information specifically for this device, i.e., this chip/package/wire-bonding combination.
- Line 14: keyword `Tester` is followed by a quoted string conforming to the suggested convention, i.e., identifying vendor, model, and configuration, in that order.
- Lines 15–16: these are two `DCSequence` blocks, each followed by a standard id and brace enclosed block. The id and ellipses represent code specified by STIL.2. Both sequences apply to the following `TestHead` statement since overrides, i.e., sequences of the same name, are not defined in the `TestHead` block.
- Lines 21–29: keyword `ChannelMap` is followed by a brace enclosed block of signal-to-tester-channel mappings. All signals shall be accounted for, i.e., all unnamed `Signals` block signals and any signals from named `Signals` blocks referenced in the `Chip` and `Device/Package` blocks.

Figure 53 shows a diagram of a dual chip package (dual site testing).

The `Device` block example in Figure 54 is for a 2 chip package with individual chips bonded directly to package pins, i.e., no wrapper.[35]

[35] If the two chips were enclosed in a hardware wrapper, they should be treated as though they were one.

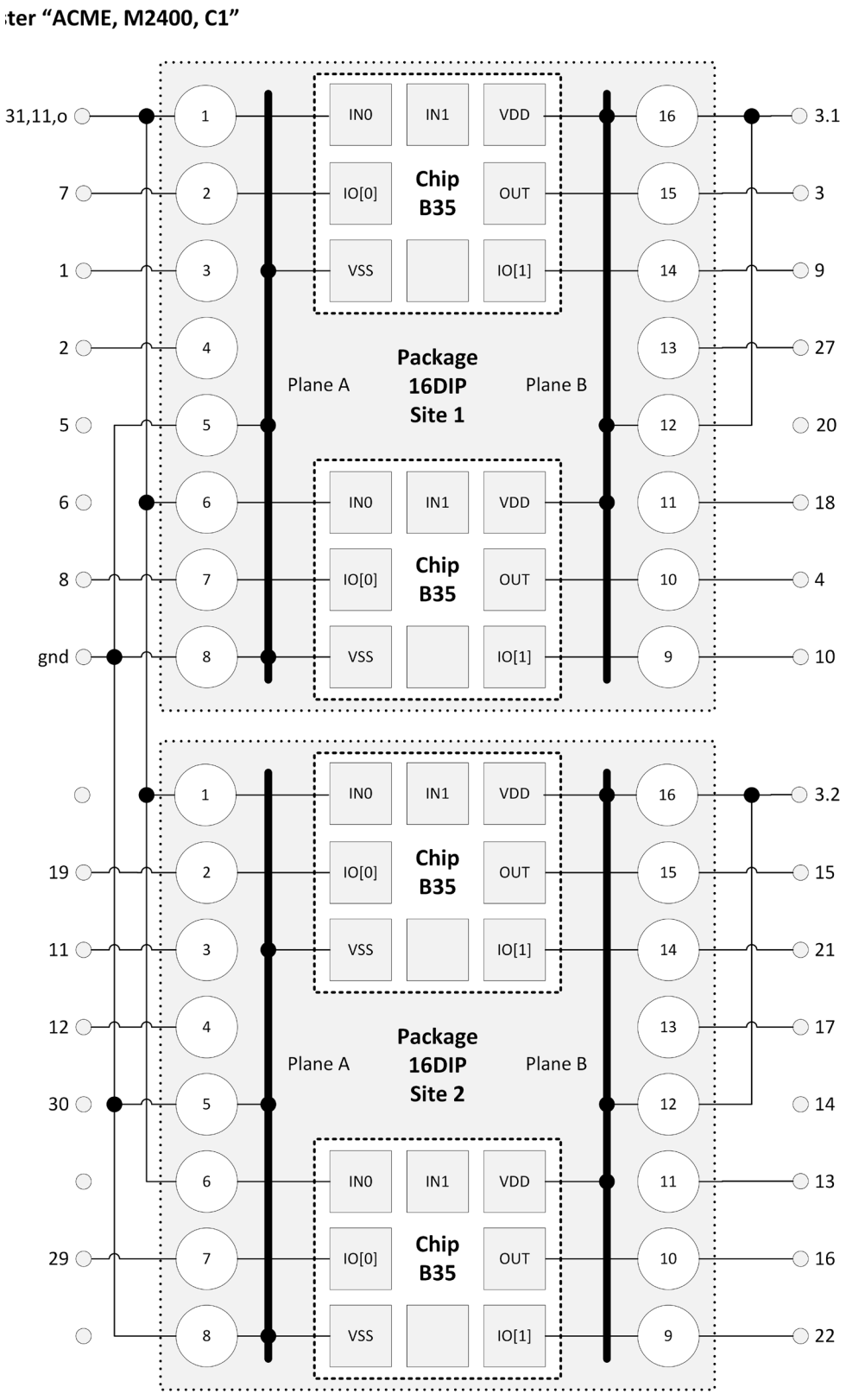

Figure 53—Diagram: dual chip package, dual site testing

```
 1 Device TWO_B35_IN_16DIP {
 2   Chip B35;          // 1st chip in package
 3   Chip B35;          // 2nd chip in package
 4   Package 16DIP;
 5   Signals      TWO_B35_IN_16DIP;
 6   SignalGroups TWO_B35_IN_16DIP;
 7   PinMap {           // Signal-to-package-pin map
 8     IN0      1,  6;  // 1st, 2nd chip associated pin numbers
 9     IO[0]    2,  7;
10     IO[1]    9, 14;
11     OUT     10, 15;
12     open3    3;
13     open4    4;
14     open11  11;
15     open13  13;
16     VDD      B;      // Maps signal to plane-name
17     VSS      A;      // Maps signal to plane-name
18   }
19   Tester "Verigy, 93000, standard_digital" {
20     DCSequence InitialSetup { // Beginning of program
21       '0s'    VDD { Apply '0V'; }
22       '1ms'   VDD { Connect Supply; }
23       '100us' VDD { Apply; }
24       '1ms'   ALLSIGS { Connect Load; Connect Driver Comparator; }
25     }
26     DCSequence EndOfProgram { // End of program
27       '0s'  'VDD+ALLSIGS' { Ramp '5ms' '0V'; }
28       '6ms' ALLSIGS { Disconnect Driver Comparator; }
29       '1ms' VDD { Disconnect Supply; }
30     }
31     TestHead {
32       Partition {
33         TestProgram TWO_B35_IN_16DIP;
34         DeviceSites 1,2 {  // Device (package) site numbers
35           ChannelMap {
36             IN0    10101, @1, @1, @1{ ChanDirection InOut; }
37             IN1    None, None, None, None;
38             OUT    [ 10103, 10104 ][ 10201, 10202 ];
39             IO[0]  [ 10105, 10106 ][ 10203, 10204 ];
40             IO[1]  [ 10107, 10108 ][ 10205, 10206 ];
41             open3  10109, 10110;
42             open4  10111, 10112;
43             open11 10113, 10114;
44             open13 10115, 10116;
45             VDD    11,13;
46             VSS    gnd;
47           }  // End ChannelMap
48         }  // End DeviceSites
49       }  // End Partition
50     }  // End TestHead
51   }  // End Tester
52 }  // End Device
```

Figure 54—Example: Device block for dual chip package, dual site testing

The following line comments refer to Figure 54:

- Line 1: keyword `Device` is followed by an id representing a chip/package/wire-bond combination.
- Lines 2–3: the two references to a previously defined chip imply that there are two chips in this device. Were there no following package reference, these two lines would not make sense because the chip names are the same.
- Lines 4–18: block refers to a previously defined package and adds information unique to this device.
- Lines 5–6: refers to previously defined signal and signal group blocks. Their contents describe signals and signal groups that are applicable to this device, i.e., chip/package/wire-bonding combination only.
- Lines 8–17: maps signals to package pins or planes. Note that signals `open3`, `open4`, `open11`, and `open13` from the named `Signals` block are mapped and that signals `VDD` and `VSS` are mapped to their respective planes.
- Line 20: keyword `Tester` is followed by a quoted string conforming to the suggested convention, i.e., identifying vendor, model, and configuration, in that order.
- Lines 21–22: these are two `DCSequence` blocks, each followed by a standard id and brace enclosed block. The id and ellipses represent code specified by STIL.2. Both sequences apply to the following `TestHead` statement since overrides, sequences of the same name, are not defined in the `TestHead` block.
- Line 23: keyword `TestHead` requires no name since there is only one. If more than one test-head is used, unique names shall be required. `TestHead` represents a physical entity.
- Line 24: keyword `Partition` is unnamed since there is only one partition. Multiple partitions divide the test-head into separately programmable entities, e.g., for a multi-project wafer. If more than one partition is used, names unique to `TestHead` shall be required. A channel may be controlled by one partition only.
- Line 25: keyword `TestProgram` is followed by a forward reference to the test program that is to be executed by this partition. Program `TWO_B35_IN_16DIP` is omitted from this example in order to focus on the device/tester interface.
- Line 26: keyword `DeviceSites` defines sites `1` and `2`. Site numbers shall be greater than zero but need not be sequential or contiguous.
- Line 27: keyword `ChannelMap` is followed by a brace enclosed block of signal-to-tester-channel mappings. All signals shall be accounted for, i.e., all unnamed `Signals` block signals and any signals from named `Signals` blocks referenced in the `Chip` and `Device/Package` blocks. Channel names used in this example represent different tester naming conventions to point out syntax usage.
- Line 28: signal `IN0` is mapped to a channel name quoted to include otherwise illegal id characters. Two channels are listed for packaged device site 1, and two for site 2. The at-sign notation indicates that all four connections are to the channel named in channel field one. Keyword `ChanDirection` is used to indicate that the tester-channel configuration does not match the signal direction, i.e., `In`, `InOut`, or `Out`. Some testers may insist that in order to perform a certain kind of test, the tester-channel configuration shall be `InOut` even though the signal is of type `In`. In this way we maintain the signal type.
- Line 29: signal `IN1` is mapped to no tester-channel, twice for site `1` and twice for site `2`.
- Line 30: signal `OUT` is mapped to one tester-channel for site `1` and another for site `2`. This line uses an alternative notation offered for packaged devices that encloses each site in brackets for easier visualization. Channel field numbers are in the same sequence they would be without brackets.
- Lines 31–32: these are mappings for signals originally defined as `IO[0..1]`.
- Lines 33–36: these are mappings for unconnected package pin signals.
- Line 37: signal `VDD` is mapped to one power supply per site.
- Line 38: signal `VSS` is mapped to the ground channel. The tester only has one. Most testers do not require this specification but it helps portability.

21. Binning

21.1 General

The conceptual model for binning comprises the following elements:

— One or more soft bin definitions (Clause 22)
— One or more hard bin definitions (Clause 23)
— One or more soft-to-hard bin maps[36] (Clause 24)
— Counters and bin related events (22.6)
— Bin setting and clearing mechanisms (Clause 33)
— A null bin called `None` (22.4)
— Bin axes (22.5)

Soft bin definitions blocks and hard bin definitions blocks shall each contain a `Pass` and a `Fail` group. Each group shall contain one or more axes. Each axis shall contain one or more related STIL.4 bins. In the 22.2 example referred to extensively throughout this document, the axis named `ClockSpeed` has bins `"3.00GHz"`, `"2.93GHz"`, and `"2.66GHz"`, and the axis named `CacheSize` has bins `"8Mb"` and `"4Mb"`.[37]

Hard bin groups shall be constrained to a single bin axis which may be named or unnamed. Soft bin groups may have one unnamed explicit or implicit bin axis, or one or more named bin axes.[38]

If a bin map is to be used, the `TestProgram` block shall refer to the bin map by name. This bin map shall in turn reference soft bin definitions and optionally hard bin definitions by name. See Clause 24 for a more detailed bin map description.

Hard and soft bins shall be cleared automatically by the on `START` event handler before the test it refers to is executed. When a bin is set, it shall remain set until cleared by the next on `START` event or by a `ClearBin`[39] function call. Hard bins shall be set according to the `BinMap` specified in the `TestProgram` block when execution is complete.

Soft bins may be set or cleared by the STIL.4 programmer by using `SetBin`, `SetBinStop`, or `ClearBin` statements in the action blocks specified in Table 21. The semantics of these functions are explained in Clause 33. Each binning function takes an argument which may refer to a single soft bin, a soft bin axis, or a soft bin group (either `Pass` or `Fail`).

21.2 Binning element reference

Syntax for referring to a single bin:

unary_bin_expr ::=
  <
    (<**Pass.** | **Fail.**>**)(BinAxes**[<BIN_AXIS_NAME | BIN_AXIS_INDEX> **].)Bins**[<BIN_NAME | BIN_INDEX >] |
    <BIN_NAME | BIN_NUMBER>
  >

[36] No more than one soft-to-hard bin map, one set of soft bin definitions, and one set of hard bin definitions per test program may be active at one time.
[37] These particular bin names are double quoted so as not to violate identifier syntax rules.
[38] The code for an explicit axis with no name is `BinAxis { ... }`.
[39] Only soft bins may be cleared via the ClearBin action.

Syntax for referring to all bins along a bin axis (related bins, e.g., speed bins):

unary_bin_axis_expr ::=
  <
    (<**Pass** | **Fail.**>)**BinAxes**[<BIN_AXIS_NAME | BIN_AXIS_INDEX>] | BIN_AXIS_NAME
  >

Syntax for referring to all bins in a group:

group ::= **<Pass | Fail>**

Syntax for referring to multiple bins, i.e., all bins in a group or along an axis:

multi_bin_expr ::= **<** *group* **|** *unary_bin_ axis_expr* **>**

22. SoftBinDefs

22.1 SoftBinDefs syntax

The general form of the `SoftBinDefs` block is as follows:

```
SoftBinDefs (BIN_DEF_NAME) {
  (StartBinNumber integer;)          // Default if unspecified is 1
  (BinNumberIncrement integer;)  // Default if unspecified is 1
  Pass {
    (Color string;)   // Pass bin default color name or hex RGB, "green" if unspecified
    (bin_definition)+ | (bin_axis_definition)+
  }   // end Pass
  Fail {
    (Color string;)   // Fail bin default color name or hex RGB, "red" if unspecified
    (bin_definition)+ | (bin_axis_definition)+
  }   // end Fail
}
```

One unnamed `SoftBinDefs` block shall be permitted. Named `SoftBinDefs` blocks shall be permitted only in FlowExtended mode. Metatypes *bin_definition* and *bin_axis_definition* are explained in 22.3 and 22.5, respectively.

Each soft and hard bin definition block contains one `Pass` and one `Fail` group. Each of these groups contains at least one axis, explicitly or implicitly, named or unnamed. If a group contains multiple axes, each axis shall be uniquely named within the group for unambiguous reference. See examples in Figure 57 and Figure 59 (in 22.2 and 23.2, respectively).

When the bin group contains bin definitions directly, a single anonymous bin axis is implied. Alternatively, bin definitions may be enclosed in a bin axis block. Hard bin definitions shall be constrained to one bin axis.

Color: optional user settable attribute which may be specified at the beginning of the block and may be used to set the default display color for all bins within the group (colors that override the default may be specified per bin). If left unspecified, the default colors are red and green for the `Fail` and `Pass` groups respectively.

Groups have a data access function which may be invoked by the following syntax:

<**Pass**|**Fail**>**.BinAxes.size():** a phrase that returns the axis count for either `Pass` or `Fail` group as type `Integer`. It may be used to establish an end point for iterating over axes. The axis count shall remain constant during program execution.

A bin's attributes (see 22.3) include a bin number. Bin numbers can be automatically generated using `StartBinNumber` for the first bin and `BinNumberIncrement` for any bin after the first. Automatic bin numbering shall be in effect only if attribute `Number` is not set on any bin. If bin attribute `Number` is set for any bin then automatic bin numbering is not in effect, and `Number` shall be set for each bin within the `SoftBinDefs` block. Each bin's number shall be unique within the `SoftBinDefs` block.

StartBinNumber: Sets the starting bin number for automatic bin number generation. If not present, the default `StartBinNumber` is 1.

BinNumberIncrement: Sets the bin number increment for automatic bin number generation. If not present, the default `BinNumberIncrement` is 1.

Figure 55 is a pictorial representation of the `Pass` group in the example of Figure 57. Each of the gray cells formed by the bin axes may eventually be mapped to a hard bin by a `BinMap` (see Clause 24). When there is only one axis like in the `Fail` group of Figure 57, the soft to hard bin mapping relationship is one to one.

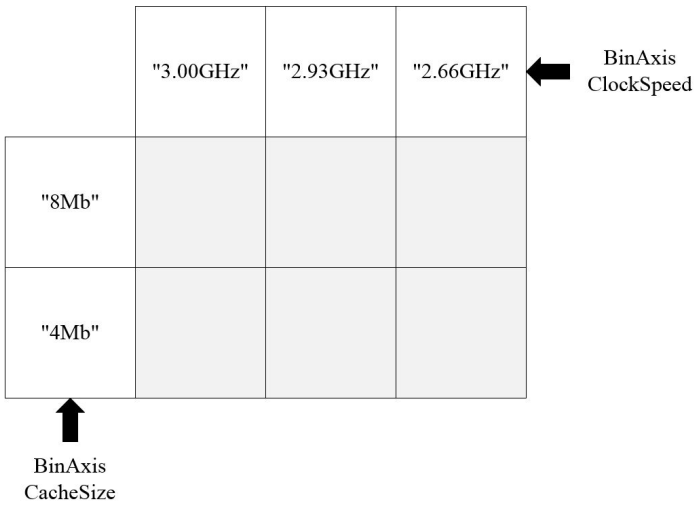

Figure 55—Diagram: pass group with two bin axes

22.2 SoftBinDefs examples

The soft bin definitions example of Figure 56 show a typical, common usage in which bin numbers are explicitly specified instead of being automatically generated, and in which there are no bin axes.:

```
1 // Simple, common usage of SoftBinDefs without StartBinNumber,
2 // BinNumberIncrement, or BinAxis
3 SoftBinDefs softbindefs {
4   Pass {
5     Bin "3.00GHz" { Number 1; Color Green; }
```

```
 6     Bin "2.93GHz" { Number 2; Color Green; }
 7     Bin "2.66GHz" { Number 3; Color Green; }
 8   }
 9   Fail {
10     Bin ContactOpens  { Number 6; Color Red; }
11     Bin ContactShorts { Number 7; Color Red; }
12     Bin Functional    { Number 8; Color Red; }
13     Bin Timing        { Number 9; Color Red; }
14     Bin Levels        { Number 10; Color Red;}
15   }
16 }
```

Figure 56—Example: soft bin definitions—simple, common usage

The soft bin definitions example of Figure 57 automatically generates default numbers for each bin, and includes an illustration of the use of bin axes. Note the difference between the bin number and a numeric bin index.

```
 1 SoftBinDefs softbindefs {
 2   StartBinNumber 1;      // Sets default start bin number
 3   BinNumberIncrement 1;  // Sets default bin number increment
 4   Pass {
 5       BinAxis ClockSpeed {
 6           Bin "3.00GHz";   // Index 0, number 1
 7           Bin "2.93GHz";   // Index 1, number 2
 8           Bin "2.66GHz";   // Index 2, number 3
 9       }
10       BinAxis CacheSize {
11           Bin "8Mb";       // Index 0, number 4
12           Bin "4Mb";       // Index 1, number 5
13       }
14   }
15   Fail {
16     Bin ContactOpens;      // Index 0, number 6
17     Bin ContactShorts;     // Index 1, number 7
18     Bin Functional;        // Index 2, number 8
19     Bin Timing;            // Index 3, number 9
20     Bin Levels;            // Index 4, number 10
21   }
22 }
```

Figure 57—Example: soft bin definitions—bin axes, autoincrementing bin numbers

22.3 Bins

A `Bin` is the lowest object in the STIL.4 binning containment hierarchy which is composed of a bin definitions block (either `SoftBinDefs` or `HardBinDefs`) which contains contains two group blocks, one named `Pass` and one named `Fail`, each of which contain one or more axes (`BinAxis`), each of which contains one or more bins. See the coding example Figure 57 in 22.2 for a better view of the hierarchy.

bin_definition ::= **Bin** < BIN_NAME ; | BIN_NAME { *bin_attributes* } >

BIN_NAME shall use meta-type *name_segment* syntax.

bin_attributes ::=
(**Color** *string*;)
(**Enable** *boolean*;)
(**Number** *signed_integer*;)
(**WafermapChar** <*simple_char* | *special_char* | "*simple_char* | *special_char* | *space*">;) // Hard bin only
(**Terse** *string*;)
(**Verbose** *string*;)

This clause enumerates user settable attributes by keyword.

Color: optional, may be used to override the display color specifcations or defaults for the `Fail` and/or `Pass` groups. The argument for **Color** is a literal, naming either the color, e.g., "red", or a 24 bit hexadecimal RGB code, e.g., "#FF0000". This literal should be quoted, but may be unquoted. It is recommended to use the same form (quoted or unquoted) throughout.

Enable: optional, may be set to `True`, the default, or `False`. Bin axis attribute `size()` is not affected.

Number: optional, specifies bin numbers on a per-bin basis, when automatic bin numbering is not used (see 22.1).

Terse: optional, may be used to store a brief bin descriptor in the form of a `String`.

Verbose: optional, may be used to store a verbose bin descriptor in the form of a `String`.

WafermapChar: may be used to store a single character suitable for printing a wafer map. Set to space character by default. The parameter may be a single visible character or a string containg a single character. The following statements shall be legal:

```
1 WafermapChar T;
2 WafermapChar "T";
3 WafermapChar " ";
```

The following line comments refer to WafermapChar statements above:

— Lines 1 and 2: both lines set the wafermap character to T.
— Line 3: this is the only way to explicitly set the wafermap character to a space.

The following example shows how to set these attributes by redefining bin `Contact` from the example in 23.2. For brevity, the definitions, group, and axis blocks are omitted:

```
1 Bin Contact {
2     Color "#00FFFF";        // Alias cyan
3     Enable False;
4     Number 6;               // Number all or none in SoftBinDefs
5     Terse "Contact";
6     Verbose "Contact opens and shorts";
7     WafermapChar O;
8 }
```

The following line comments refer to bin `Contact` statements above:

— Line 2: specifies GUI color via string containing a hexadecimal value, could have specified the same color via string "cyan".
— Line 3: the bin is disabled. Tests and/or flow-nodes can react to this state via bin member function `isEnabled()`.

— Line 4: bin number is specified. This number shall be unique within the bin definitions block. Bin numbers shall be specified for all bins or none. When bin numbers are unspecified, bins are automatically numbered in order of appearance starting with 1 by default (see bin definitions attribute `StartBinNumber`).

Bins also have data access functions which may be accessed via the following format:

unary_bin_expr.*bin_data_access_function*

Meta-type *unary_bin_expr* is described in 21.2. Bin data access functions represented by meta-type *bin_data_access_function*:

index(): returns the bin index on the axis where the it resides in the form of an `Integer`.

isEnabled(): returns `True` if the bin is enabled, `False` otherwise.

getBinGroup(): returns enumerated type `BinGroup` value (`FAIL`, `PASS`, or `NONE`).

isSet(): returns `True` if the bin is set, `False` otherwise.

name(): returns the bin name in the form of type `String` (does not include quotes if any).

number(): returns the bin number in the form of type `Integer`.

terse(): returns the terse bin description in the form of a `String`.

verbose(): returns the verbose bin description in the form of a `String`.

wafermapChar(): returns the bin associated wafer map character in the form of a single character `String`.

22.4 Bin None (FlowExtended)

In `FlowExtended` mode, soft bin `None` is automatically defined by STIL.4. User statements defining this bin shall be neither required nor permitted. When mapping soft to hard bins, the mapping for soft bin `None` is triggered when no other soft bins have been set, regardless of whether bin `None` has been set or not.

Bin `None` data access functions may be accessed only via parameters, e.g., `TestBase` parameter `failBin` or `passBin`. They cannot be accessed directly, via e.g., `None.isSet()`, because keyword `None` assumes context-dependent semantics. Bin `None` may be used directly in unambiguous contexts such as statement `SetBin None`. See Table 17.

Table 17—Bin None standard attributes and data access functions

Attribute	Function	Value
	countSince(*counter_reset_event*)	Integer `0`, incremented when set, may be cleared by user
Enable	isEnabled()	Boolean `True`
Index	index()	Integer `0`
	getBinGroup()	Enum `BinGroup::NONE`
	isSet()	Boolean `False`, becomes `True` when set
Name	name()	String `"None"`
Number	number()	Integer `-1`
Terse	terse()	String `"NoBin"`
Verbose	verbose()	String `"NoBin"`
WafermapChar	wafermapChar()	String `" "` (single space character)

22.5 Bin axes

Each `Pass` and `Fail` group contains at least one axis, explicit or implicit, named or unnamed. If a group contains multiple axes, each axis shall be uniquely named within the group. Each axis shall contain at least one element of type `Bin`.

bin_axis_definition ::=
  BinAxis (BIN_AXIS_NAME) {
    (**MapBinLowest** | **MapBinHighest**)
    (*bin_definition*)+
  }

Metatype *bin_definition* is explained in 22.3.

MapBinLowest or **MapBinHighest**: used for bin arbitration. These keywords affect soft to hard bin mapping as follows: when mapping soft to hard bins in the `BinMap` and two or more bins on a bin axis are set, the soft bin with either the lowest or highest index is mapped to the hard bin. The default is `MapBinHighest`, commensurate with the most stringent device spec being associated with bin index 0. In this scheme, device specs are expected to relax with each index increment as is the case in the example of Figure 57.

Axis data access functions:

axis_name.**Bins.size()**: a phrase that returns the bin count for the axis and is useful for establishing a bin iteration end point. The return value is of type `Integer`. The bin count shall remain constant during program execution.

axis_name.**name()**: returns the name of the axis in the form of type `String`. The name shall remain constant during program execution.

22.6 countSince functions (FlowExtended)

In `FlowExtended` mode, each soft bin has a set of counters. The set consists of a counter for each *counter_reset_event*.

counter_reset_event [40] ::= < LOAD | LOT_START | WAFER_START | START >

During the execution of a test or flow triggered by a *counter_reset_event*, soft bin counts associated with that *counter_reset_event* are incremented each time a `SetBin` or `SetBinStop` action is executed for the associated soft bin, and are reset to zero each time a `ClearBin` action is executed for the associated soft bin.

At the termination of the test or flow triggered by a START event, soft bin counters associated with counter_reset_events other than START are incremented by one if the associated soft bin is set at that time.

Counters corresponding to a *counter_reset_event* are reset to zero each time that *counter_reset_event* occurs, before the *counter_reset_event* associated test or flow is executed.

Function `countSince` applies to soft bins, soft bin axes, and soft bin groups, and returns an `Integer`.

<**Pass**|**Fail**>**.countSince(***counter_reset_event***)**:

Returns the number of bins that have been set in the `Pass` or `Fail` group since the *counter_reset_event*.

*axis_name***.countSince(***counter_reset_event***)**:

Returns the number of bins that have been set on the axis since the *counter_reset_event*.

*unary_bin_expr***.countSince(***counter_reset_event***)**:

Returns the number of times the bin has been set since the *counter_reset_event*.

23. HardBinDefs

23.1 HardBinDefs syntax

Hard bin definitions block syntax is identical to soft bin definitions block syntax with the following exceptions:

- The keyword introducing the block is **HardBinDefs**.
- The `Pass` and `Fail` group blocks shall each be restricted to one axis, explicit or implicit, named or unnamed.

```
HardBinDefs (BIN_DEF_NAME) {
  (StartBinNumber integer;)          // Default if unspecified is 1
  (BinNumberIncrement integer;)  // Default if unspecified is 1
  Pass {
    (Color string;)   // Pass bin default color name or hex RGB, "green" if unspecified
    (bin_definition)+ | (bin_axis_definition)
  }   // end Pass
  Fail {
    (Color string;)   // Fail bin default color name or hex RGB, "red" if unspecified
    (bin_definition)+ | (bin_axis_definition)
```

[40] This is a subset of enumerated type `AsynchronousEvent` (35.1).

```
  }  // end Fail
}
```

One unnamed `HardBinDefs` block shall be permitted. Named `HardBinDefs` blocks shall be permitted only in FlowExtended mode. Metatypes *bin_definition* and *bin_axis_definition* are explained in 22.3 and 22.5, respectively.

23.2 HardBinDefs examples

The hard bin definitions example of Figure 58 show a typical, common usage in which bin numbers are explicitly specified instead of being automatically generated, and in which the hard bin color is also specified.

```
 1 HardBinDefs {
 2   Pass {
 3     Bin "3.00GHz" { Number 1; Color Green; }
 4     Bin "2.66GHz" { Number 2; Color Green; }
 5   }
 6   Fail {
 7     Bin failContact { Number 5; Color Red; }
 8     Bin dcFails { Number 6; Color Red; }
 9     Bin acFails { Number 7; Color Red; }
10   }
11 }
```

Figure 58—Example: hard bin definitions—simple, common usage

The hard bin definitions example of Figure 59 automatically generates default numbers for each bin. The following hard bin definitions example provides the basis for other examples:

```
 1 HardBinDefs hardbindefs {
 2   Pass {
 3     Bin "3.00GHz/2.93GHz/8Mb";  // Index 0, number 1
 4     Bin "3.00GHz/2.93GHz/4Mb";  // Index 1, number 2
 5     Bin Unmarketable;           // Index 2, number 3
 6     Bin Unclassifyable;         // Index 3, number 4
 7   }
 8   Fail {
 9     Bin Contact;                // Index 0, number 5
10     Bin LooseFunct;             // Index 1, number 6
11     Bin Timing;                 // Index 2, number 7
12     Bin Levels;                 // Index 3, number 8
13   }
14 }
```

Figure 59—Example: hard Bin definitions—autoincrementing Bin numbers

23.3 Bins

Syntax is described in 22.3.

24. BinMap

24.1 General

Block type `BinMap` refers to soft and optionally to hard bin definitions. Hard bins shall be set according to the `BinMap` specified in the `TestProgram` block when execution is complete. This is the sole mechanism for setting hard bins.

The following rules apply:

- When there is a single axis in the group, each mapping statement maps a soft bin to a hard bin. Two or more soft bins may map to the same hard bin.
- When there are two or more axes in a group, each mapping statement shall map a cell of the resulting array to a hard bin. Multiple soft bin combinations may map to the same hard bin.
- Each combination of bins, one from each axis contained within a group,[41] shall have a corresponding mapping statement. A mapping statement is triggered after all its soft bins are set. `Pass` soft bins or soft bin combinations shall be mapped only to `Pass` hard bins, and `Fail` soft bins shall be mapped only to `Fail` hard bins. In `FlowExtended` mode, one statement mapping soft bin `None`[42] to a hard bin is permitted. This mapping is triggered after no other soft bin is set.

24.2 BinMap syntax

The syntax for specifying a `BinMap` is as follows:

(**BinMap** BIN_MAP_NAME {
  (**SoftBinDefs** BIN_DEF_NAME;)
  (**HardBinDefs** BIN_DEF_NAME;)
  *bin_map_stmt**
})*

SoftBinDefs and **HardBinDefs** specifiers shall be allowed only in `FlowExtended` mode. A bin map that has no `SoftBinDefs` or `HardBinDefs` references shall use the unnamed `SoftBinDefs` or `HardBinDefs` blocks.

bin_map_stmt ::=
  Map *soft_bin_expr*+ **->** *hard_bin_expr*;

Multiple instances of meta-type *soft_bin_expr* shall be separated by white space.

soft_bin_expr ::= *unary_bin_expr;*
hard_bin_expr ::= *unary_bin_expr;*

Meta-type *soft_bin_expr* shall be from a `SoftBinDefs` block. Meta-type *hard_bin_expr* shall be from a `HardBinDefs` block. Meta-type *unary_bin_expr* is described in 21.2.

[41] `BinAxis` keywords `MapBinLowest` and `MapBinHighest` control which of multiple set bins on an axis is used for mapping.
[42] In `FlowExtended` mode, soft bin `None` is automatically defined by STIL.4.

24.3 BinMap example

Figure 60 shows a **BinMap** example showing a simple usage, with unnamed **SoftBinDefs** and **HardBinDefs** blocks (hence, the **BinMap** need not specify them), and a simple, flat **BinMap**.

```
BinMap binmap {
 "3.00GHz" -> "3.00GHz";
 "2.93GHz" -> "2.66GHz";
 "2.66GHz" -> "2.66GHz";
 ContactOpens -> failContact;
 ContactShorts -> failContact;
 Functional -> acFails;
 Timing -> acFails;
 Levels -> dcFails;
}
```

Figure 60—Example: BinMap using unnamed SoftBinDefs, HardBinDefs

Figure 61 shows another **BinMap** example, commensurate with the soft bin definition example in 22.2 and the hard bin definitions example in 23.2.

```
BinMap binmap {
  SoftBinDefs softbindefs;
  HardBinDefs hardbindefs;

  Map None[43]              -> 4;  // Unclassifyable

  Map "3.00GHz"  "8Mb"  -> 1;  // Pass "3.00GHz/2.93GHz/8Mb"
  Map "2.93GHz"  "8Mb"  -> 1;  // Pass "3.00GHz/2.93GHz/8Mb"
  Map "2.66GHz"  "8Mb"  -> 3;  // Pass Unmarketable
  Map "3.00GHz"  "4Mb"  -> 2;  // Pass "3.00GHz/2.93GHz/4Mb"
  Map "2.93GHz"  "4Mb"  -> 2;  // Pass "3.00GHz/2.93GHz/4Mb"
  Map "2.66GHz"  "4Mb"  -> 3;  // Pass Unmarketable

  Map ContactOpens      -> 5;  // Fail ContactOpens
  Map ContactShorts     -> 5;  // Fail ContactShorts
  Map Functional        -> 6;  // Fail LooseFunct
  Map Timing            -> 7;  // Fail Timing
  Map Levels            -> 8;  // Fail Levels
}
```

Figure 61— Example: BinMap using named SoftBinDefs, HardBinDefs

25. Flow conceptual model

The flow conceptual model in Figure 62 describes the interaction between the `Flow`, `FlowNode`, `Test`, `TestProgram`, and binning syntax blocks.

[43] Bin `None` usage requires `FlowExtended` mode.

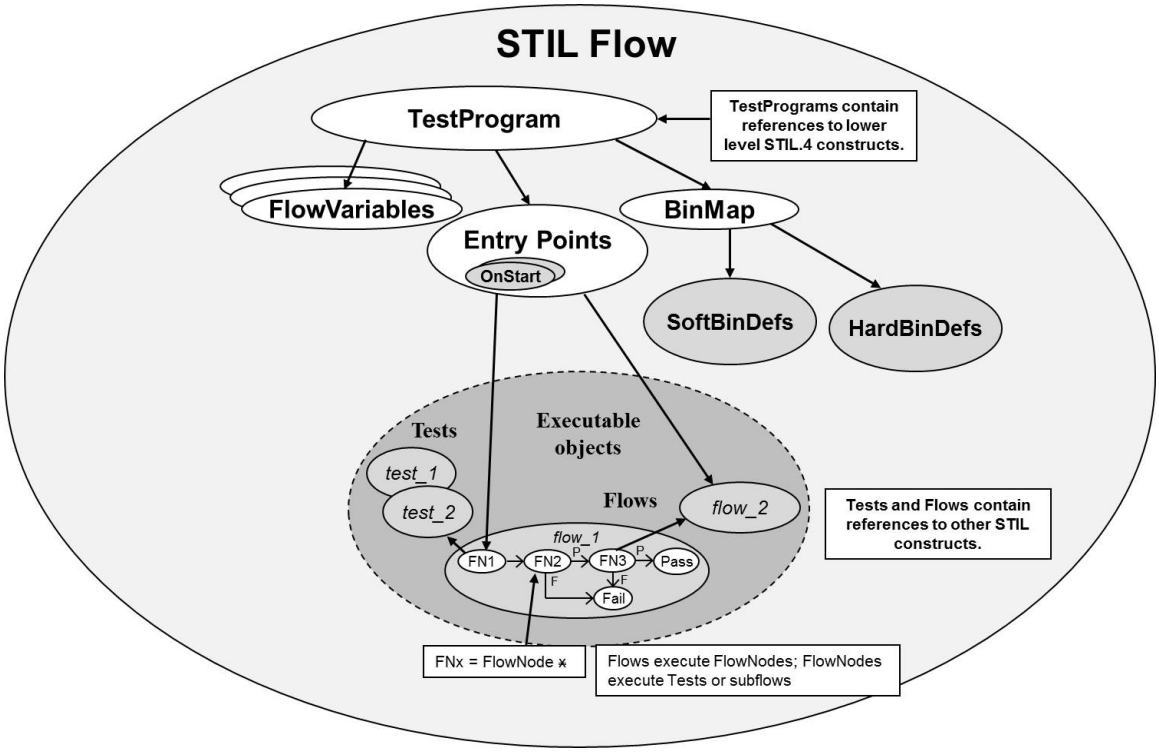

Figure 62—Diagram: STIL.4 conceptual model

An entry-point, described in 34.4, references a single `Test` or `Flow` which it executes when it's associated asynchronous event, such as the push of a tester start button, triggers it. Every execution path can be traced back to an entry-point.

Figure 63 illustrates a `Flow` block. A `Flow` contains a collection of `FlowNode` blocks. The first `FlowNode` as it appears syntactically in the `Flow` is the start point. A `Flow` may be invoked by an entry-point or a `FlowNode`.

Figure 64 illustrates a `Test` block. A `Test` may contain a set of parameter assignments and reference a `TestMethod`. The `TestMethod` is not defined as part of STIL.4.

Figure 65 shows multiple `FlowNode` objects of which one is shown in detail. Each `FlowNode` refers to a single `Test` or `Flow`. A `Test` may be referred to by zero or more `FlowNode` objects. A `FlowNode` has one entry point and one or more exit ports. An exit-port may stop the execution flow via a stop or bin and stop action, or direct the continuation. The execution flow may continue with another `FlowNode` within the parent `Flow` or return to the caller of the parent `Flow`. A bin assignment may be made without stopping the execution flow.

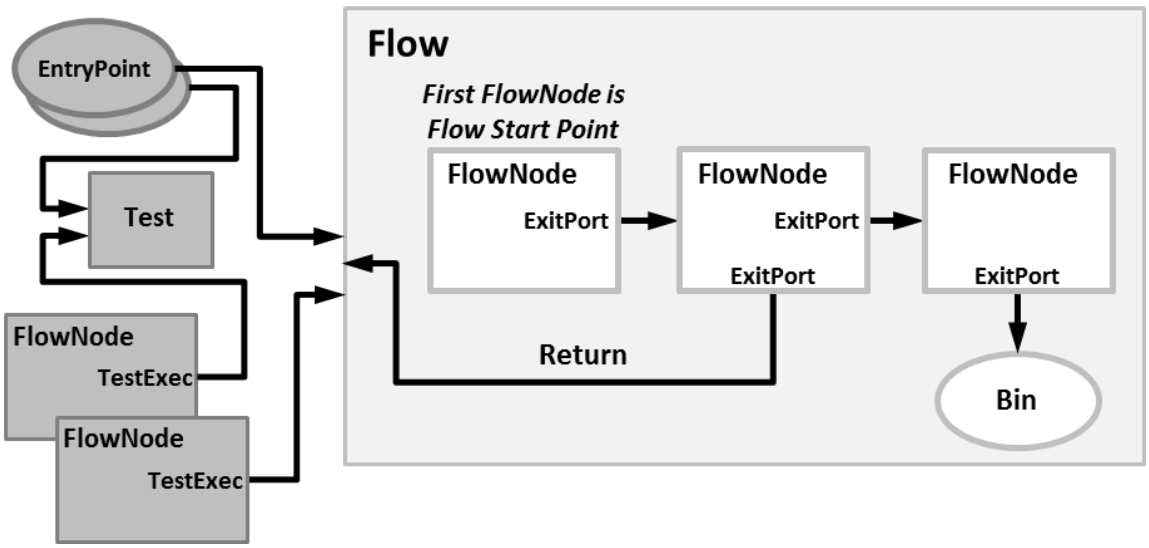

Figure 63—Diagram: conceptual model of flow

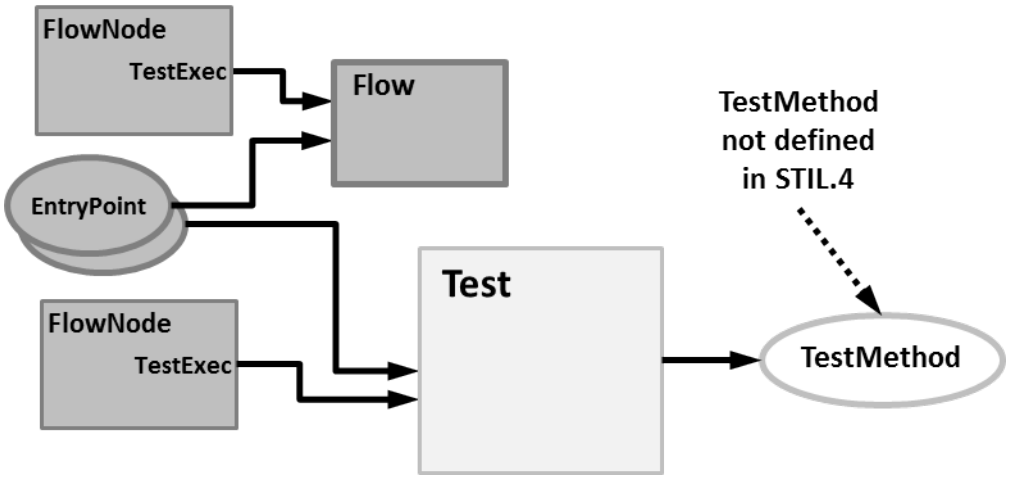

Figure 64—Diagram: conceptual model of test

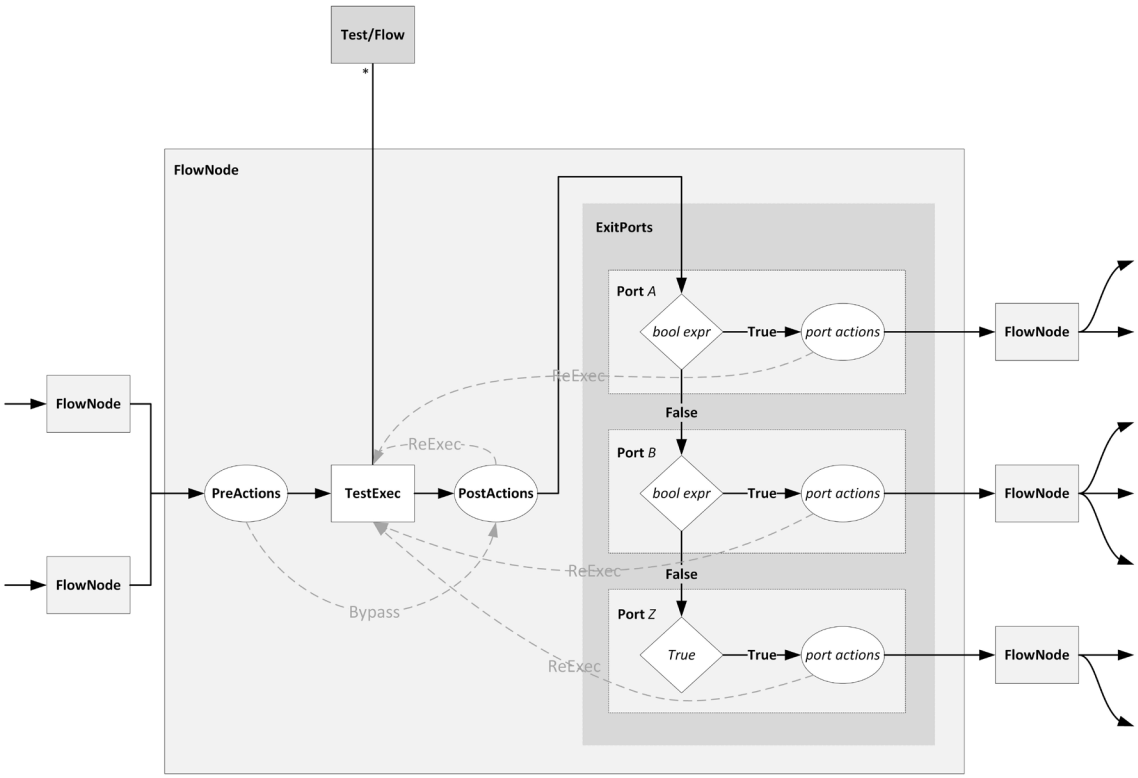

Figure 65—Diagram: conceptual model for flow node

26. Flow conceptual model (FlowExtended)

26.1 General

This conceptual model is the foundation for `FlowExtended` syntax and may be used to design a STIL.4 database. A STIL.4 compliant tool is not required to adhere to the model however it should behave as though it did. Regardless of syntactical shortcuts, the underlying conceptual model remains the same.

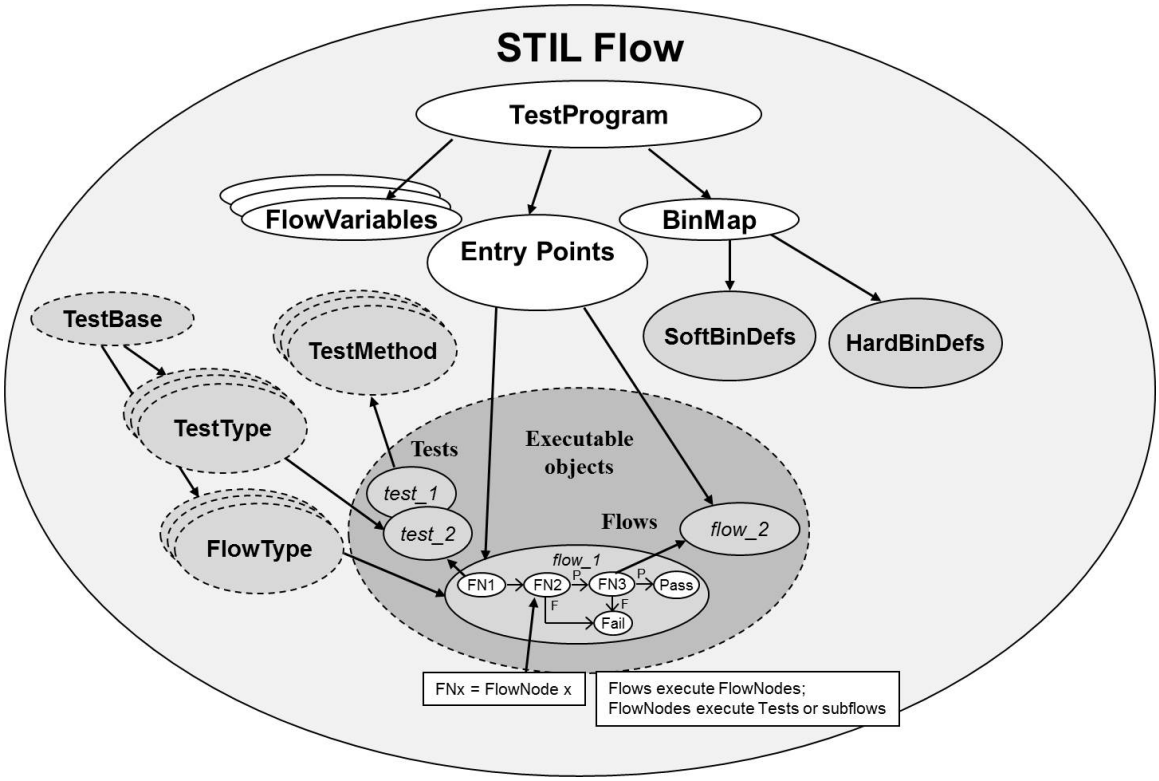

Figure 66—Diagram: STIL.4 conceptual model (FlowExtended)

The STIL.4 components describing the test-flow execution sequence are entry-points, tests, and flow-nodes. Each component is a separate object.

An entry-point, covered in detail under 34.4, refers to a single test or flow which it executes when it's associated asynchronous event, such as the push of a tester start button, triggers it. Every execution path can be traced back to an entry-point. When a test or flow is executed, it is by way of an entry-point or a flow-node. A test or flow object can be created by instantiating a test-type or flow-type respectively, or using keyword `TestMethod` to refer to a test-method by name and parameter list (see 29.2). All `FlowExtended` test and flow objects shall support `TestBase` properties (see 35.3.3).

Figure 67 shows a test or flow. Tests and flows are so similar that they can be used interchangeably. A test or flow has one entry point and one exit point. It may pass or fail. The ovals contain commands whose primary function is to control flow, directly, or indirectly via variables. The ovals represent virtual member functions, i.e., functions that may be overridden when one test-type or flow-type is derived from another (inheritance). Actions are covered in Clause 33. The rectangle labeled `TestExec` embodies the intrinsic difference between different kinds of tests and flows.

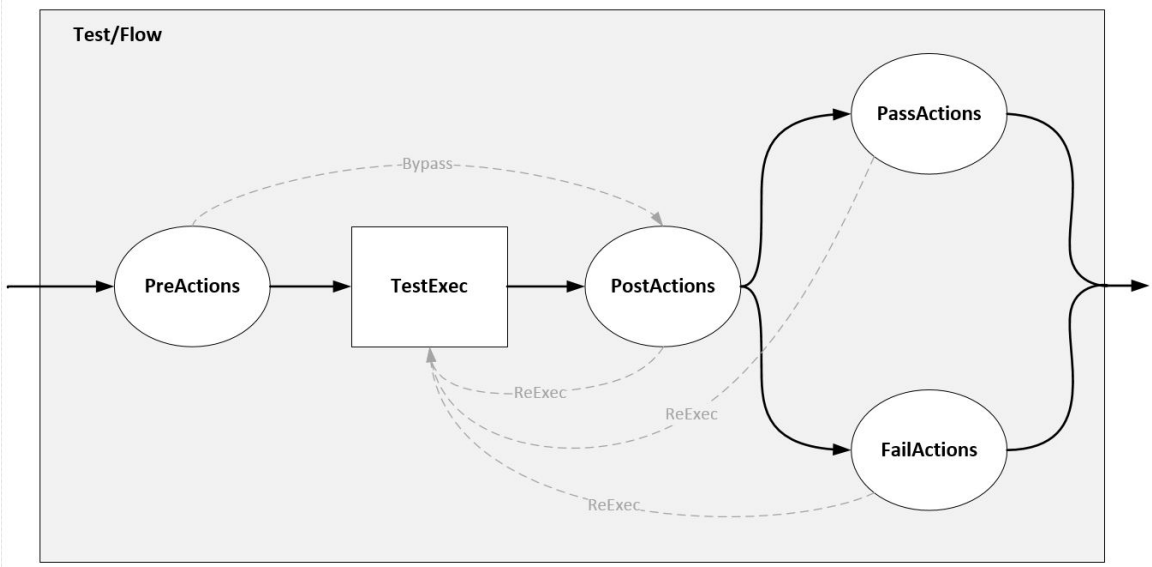

Figure 67—Diagram: conceptual model for test and flow (FlowExtended)

Type `TestBase` is at the root of every test-type or flow-type inheritance hierarchy. It is an abstract type, meaning that unlike the types derived from it, it cannot be instantiated. Type `TestBase` embodies the common denominator between all tests and flows. That includes parameters and variables not shown in Figure 63. See 35.3.3 for a more concise description of `TestBase`.

Figure 68 shows multiple flow-nodes of which one is shown in detail. Flow-nodes occur within the `TestExec` block of flows and tests instantiated from test-types that contain flow-nodes. Each flow-node refers to a single test or flow. The test or flow is executed when the flow-node is triggered. A test or flow may be referred to by zero or more flow-nodes. A flow-node has one entry point and one or more exit ports. An exit-port directs the flow to another flow-node or to the post-actions of the test or flow that contains the flow-node. Flow-node connections are constrained to within the test or flow where the flow-nodes appear. STIL.4 defines a standard flow-node that has a pass and a fail port for the purpose of allowing syntactical shortcuts (see 35.3).

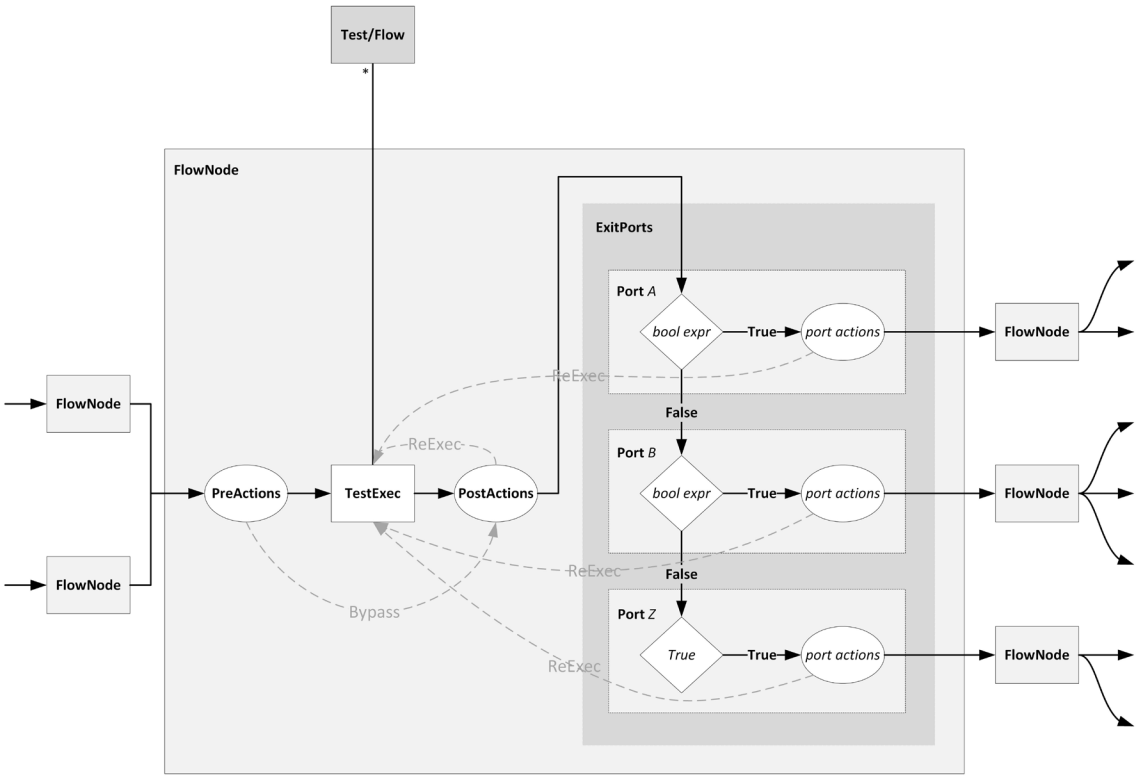

Figure 68—Diagram: conceptual model for flow node

26.2 Flow-related types

This clause describes the form of flow related types, i.e., `TestBase`, `TestType`, `FlowType`, and `FlowNode`, and provides some usage examples. STIL.4 also defines standard components based on the syntax for these types. These are described under Clause 35.

26.3 Inheritance

STIL.4 inheritance applies to test-types and flow-types with some constraints. Only single inheritance is supported. A flow-type shall inherit from `TestBase` or another flow-type. A test-type shall inherit from `TestBase`, or another test-type or flow-type. There is no constraint on the length of the inheritance chain. Every inheritance chain ultimately traces back to `TestBase` as its most distant base type. All test-types and flow-types may be used as base types.

All data is inherited cumulatively, i.e., the most distant derived type contains the sum of all the data in the inheritance hierarchy. The `Parameter` and `FlowVariables` blocks contain all the data. Parameters are public; therefore, the most distant derived type has access to the parameters of any of its ancestors as well as its own. Variables, on the other hand, are private and therefore accessible only to the type containing the `FlowVariables` block. The test/flow-type name space contains all the parameters in the inheritance hierarchy plus the variables defined in its own local `FlowVariables` block.

On inheritance, the derived test-type or flow-type may override default parameter initialization values and/or narrow constraints imposed by the base test or flow via parameter attributes. The following attribute overrides are permitted:

- For all types, `Required` may override `Optional`.
- For type `General`, attribute `Units` may override unspecified units.
- For type `BinSpec`, `BinType Pass`, or `Fail` may override a `BinType` with `Pass`/`Fail` unspecified.
- For type `BinSpec`, `BinType Bin`, `Axis`, or `Group` may override a `BinType` with `Bin`, `Axis`, or `Group` unspecified or to narrow a combination specified in the base.

For illustrative purposes only, the example in Figure 69 shows a pared-down version of `TestBase` and test-types derived from it.

```
// ------------------------------------------------------------------
TestBase {
  Parameters {  // failBin potentially affected by inheritance
      InOut BinSpec failBin = None { Optional; BinType Fail; }
  }
}
// ------------------------------------------------------------------
TestType DerivedTypeLegalA {
  Inherit TestBase { // override failBin default from None to Failed
       failBin = Failed;
  }
}
// Reaffirm TestBase failBin value (None), constrain to BinType Fail
TestType DerivedTypeLegalB {
  Inherit TestBase {
       failBin = None { BinType Fail, Group; }
  }
}
// Accept TestBase failBin value (None), constrain to BinType Fail
TestType DerivedTypeLegalC {
  Inherit TestBase {
       failBin { BinType Fail, Group; }
  }
}
// Override TestBase failBin value to Failed. Constrain to Group.
TestType DerivedTypeLegalD {
  Inherit TestBase {
       failBin = Failed { BinType Group; }
  }
}
```

Figure 69—Example: inheritance with overrides

The derived type shall inherit member functions `PreActions`, `TestExec`, `PostActions`, `PassActions`, and `FailActions` from the base-type. The derived type may override each individual base-type function in aggregate by defining it locally, e.g., one cannot inherit or override part of the `FailActions` block.

26.4 Instantiation and execution

A test may be instantiated from a test-type at the top[44] or `TestProgram`[45] level and later referred to by name at the point of use, or it may be instantiated inline anonymously at the point of use. Top-level instance statements shall not be permitted when the input stream contains more than one `TestProgram` block. Point of use is either at an entry-point or a flow-node's `TestExec` via a *reference_stmt*:

reference_stmt ::= < *test_reference_stmt* | *flow_reference_stmt* >

test_reference_stmt ::=
    < TEST_NAME ; | // Reference to predefined test by name
      TEST_TYPE; | // Reference to test created inline from TestType using defaults
      TEST_TYPE { *param_override_stmts* } // Reference to test created inline from TestType
    >

flow_reference_stmt ::=
    < FLOW_NAME; | // Reference to predefined flow by name
      FLOW_TYPE ; | // Reference to flow created inline from FlowType using defaults
      FLOW_TYPE { *flow_init_stmts* } // Reference to flow created inline from FlowType
    >

27. TestBase definition (FlowExtended)

Keyword `TestBase` represents an abstract type. In other words, an instance of `TestBase` cannot be created; however, `TestBase` is at the root of every test or flow inheritance hierarchy and therefore represents the common denominator of every test-type and flow-type.

27.1 TestBase syntax

The general form of a `TestBase` definition is as follows:

TestBase {
  (**Parameters** {
    ((<**In** | **Out** | **InOut**>) *param_elements_stmt*)*
  })
  (**FlowVariables** { (*var_elements_stmt*)* })
  (**PreActions** { (*action_stmt*)* })
  (**TestExec**;)
  (**PostActions** { (*action_stmt*)* })
  (**PassActions** { (*action_stmt*)* })
  (**FailActions** { (*action_stmt*)* })
}

Other than the fact that `TestBase` has no `Inherit` statement, `TestBase` block body syntax is the same as `FlowType` body syntax. `TestType` body syntax differs only at the `TestExec` line. `TestType` and `FlowType` shall have `TestBase` as their most distant ancestor in any inheritance chain.

[44] When a test is instantiated at the top level, its parameters and variables may refer only to globally visible data, i.e., data defined in unnamed blocks.
[45] Tests may always be instantiated within the `TestProgram` block where they can access globally visible data plus data brought into scope by named block references within the `TestProgram` block.

param_elements_stmt ::= <
param_type param_definition_stmt |
param_type { (*param_definition_stmt*)+ }
>

param_type and *stil_object_expr* are defined in 6.11.

value_expr is defined in 17.2.

param_value_expr ::= *value_expr* | *stil_object_expr*

For `In` and `Out` Parameters, the following definitions of *param_definition_stmt* apply:

param_definition_stmt ::= <
// Uninitialized scalar variable, optional type matching *units* allowed for *real_var_type* only
PARAM_NAME (= **None**(*units*)); |
PARAM_NAME (= **None**(*units*)) { (*param_attributes*)* } |

// Initialized scalar variable.
PARAM_NAME = *param_value_expr*; |
PARAM_NAME = *param_value_expr* { (*param_attributes*)* } |

// Unintialized array variable
// Optional type matching *units* allowed for *real_var_type* only
PARAM_NAME[*int_expr*] (= **None**(*units*)); |
PARAM_NAME[*int_expr*] (= **None**(*units*)) { (*param_attributes*)* } |

// Initialize array elements to distinct values
PARAM_NAME[(*int_expr*)] = [*value_list*]; |
PARAM_NAME[(*int_expr*)] = [*value_list*] { (*param_attributes*)* } |

// Initialize all array elements to same value
PARAM_NAME[*int_expr*] = *value_expr*; |
PARAM_NAME[*int_expr*] = *value_expr* { (*param_attributes*)* }
>

For `InOut` Parameters, the following definitions of *param_definition_stmt* apply:

param_definition_stmt ::= <
// Uninitialized scalar variable, optional type matching *units* allowed for *real_var_type* only
PARAM_NAME (= **None**(*units*)); |
PARAM_NAME (= **None**(*units*)) { (*param_attributes*)* } |

// Initialized scalar variable.
PARAM_NAME = &*stil_object_expr*; |
PARAM_NAME = &*stil_object_expr* { (*param_attributes*)* } |
>

param_attributes ::= <
Description *string*; | // Default is empty string, i.e. ""
ReInitAt ASYNC_EVENT_NAME; |
Units "*units_expr*"; | // For type General with initial value of None
< Optional | **Required** >
>// Optional test and flow parameters set to None are to be ignored.

Every test-type and flow-type inherits the capability to define an optional `Parameters`, `FlowVariables`, `PreActions`, `TestExec`, `PostActions`, `PassActions`, and/or `FailActions` block. Of these, `Parameters` and `FlowVariables` represent data. `PreActions`, `TestExec`, `PostActions`, `PassActions`, and `FailActions` represent protected virtual functions, i.e., functions that are not publicly accessible but may each be replaced by functions of the same name in the derived type. Once a test-type or flow-type is instantiated, its parameters and variables are initialized. Once an entry-point is triggered, it automatically executes the functions of its test or flow in the order in which they are required to appear in the test or flow definition except that only one of either `PassActions` or `FailActions` shall be executed.

A parameter or variable defined in the local `Parameters` or `FlowVariables` blocks respectively may hide a variable of the same name defined in the unnamed global `FlowVariables` block and any of the named `FlowVariables` blocks referenced in the `TestProgram` block.

Parameters: this block contains publicly accessible variables. Once a test or flow is instantiated, its parameters may be accessed via dot notation, e.g., expression `testname.parametername`. An `InOut` parameter, i.e., a reference, may be reassigned. The difference between calling a function and executing a test or flow is that a function 'forgets' its parameters on exit whereas a test or flow does not. In the type definition, each parameter shall be initialized to a default value, explicitly or implicitly. In the absence of an explicit default value, the parameter is initialized to value `None`, a value that indicates lack of initialization and may be detected via operators == or !=. At test or flow instantiation, the value `None` shall be legal only for parameters with attribute `Optional`. A required parameter's default value may be `None` but the instantiation value shall be other than `None`. A test-type or flow-type collects all the parameters in its inheritance hierarchy in one namespace.

NOTE—One parameter may refer to another from the same test-type or flow-type definition, or test or flow instantiation, but define before use rules apply. At instantiation, parameters are initialized in the order they appear in the test-type definitions' parameter block. When a test or flow is instantiated, parameter initialization statements shall be specified in test-type or flow-type definition parameter order but optional parameters may be omitted. With regard to inheritance hierarchies, `TestBase` parameters are followed by the derived test or flow parameters in order of derivation.

Parameter declaration syntax is identical to variable declaration with one exception. Parameters have an additional modifier specifying directionality, i.e., either `In`, `Out`, or `InOut` for input, output, or both, respectively. Semantics for the directionality modifiers are shown in Table 18.

Table 18—Parameter directionality semantics

Parameter	In	InOut	Out
Usage	Test or flow input only.	Test or flow input, output, or both.	Test or flow output only.
User settable at instantiation	Yes.	Yes.	No. Initialization value is defined by test-type or flow-type.

(Table continues)

Table 18—Parameter directionality semantics

Parameter	In	InOut	Out
Interface semantics	1. Can be changed inside test or flow, but changes are not reflected outside test or flow. 2. External access is read only	1. A reference to an object which exists outside the test or flow. 2. Changes to value(s) are reflected in the referenced object. 3. Can be re-assigned inside test or flow to reference a different external object of the same type. 4. The right-hand side (RHS) of initialization statements (initial declarations or overrides) shall consist of a single object name prefixed by the '&' operator. 5. Assignment statements modify the value(s) of the referenced object unless the RHS consists of a single object name prefixed by the address-of operator '&', in which case the reference is re-assigned. 6. External access is read or modify.	1. Object cannot be changed outside test or flow. 2. External access is read only.
With attribute **Optional**	The parameter shall be initialized to an expression whose type is consistent with the parameter type including `None`, explicitly or by omission (assigns default).	The parameter shall be initialized to the name of an instantiation whose type is consistent with the parameter type including `None`, explicitly or by omission which assigns the default.	The parameter shall be initialized to the default value defined in the TestType or FlowType. `None` shall be a legal value.
With default attribute **Required**	The parameter shall be initialized to any expression consistent with the parameter type, i.e., the default value shall be overridden by a value other than `None`.	Required by definition: the parameter shall be initialized to the name of an instantiation of a type consistent with the parameter type. `None` is invalid.	The parameter shall be initialized to the default value defined in TestType or FlowType. Test or flow output shall be a value other than `None`.

When an In or Out parameter with a Const attribute is instantiated with an expression, the right-hand side is replaced with its value at the time of instantiation.

When an InOut parameter with a Const attribute is instantiated, the test or flow cannot change the object referenced by the parameter; however, the parameter can be changed to refer to a different object of the same type

FlowVariables: this block contains variables with private access, i.e., by the test or flow containing the `FlowVariables` block only. A derived test or flow does not have access to any variables in the inheritance hierarchy but its own. `FlowVariables` use the same namespace as `Parameters`. A test-

type or flow-type adds only its own variables to the local namespace. Variable initialization begins once parameter initialization is completed. One variable may refer to another occurring previously in the same block or to a parameter of the same test or flow.

Action syntax is designed to control flow by manipulating variable contents and setting bins and either stop or continue. Actions may be used for side effect but they are specifically not used to load timing, levels, or patterns, activities performed under the purview of the `TestExec` function which generally differs from one test-type to another. Actions are covered in greater detail in Clause 33.

PreActions: this block contains actions carried out immediately before `TestExec`. Base test-type or flow-type pre-actions may be overridden in aggregate by defining a derived test-type or flow-type `PreActions` block.

TestExec: represents test or flow execution, a no-op for the standard `TestBase`. For derived types, `TestExec` followed by a semicolon represents a reference to the base type's `TestExec`[46] or a standard or local library test-type's non-STIL.4 code, the part of the code written a language other than STIL.4 which implements a standard or tester specific test description.

PostActions: this block contains actions carried out immediately after `TestExec`. Base type `PostActions` may be overridden in aggregate by defining a derived test-type or flow-type block of the same name.

PassActions: this block contains actions carried out immediately after `PostActions` if this test or flow passes. Base type `PassActions` may be overridden in aggregate by defining a derived test-type or flow-type block of the same name. Subclause 35.3.3 describes what constitutes pass.

FailActions: this block contains actions carried out immediately after `PostActions` if this test or flow fails. Base type `FailActions` may be overridden in aggregate by defining a derived test-type or flow-type block of the same name. Subclause 35.3.3 describes what constitutes fail.

27.2 TestBase example

In basic mode (see Clause 7), the `TestBase` definition shall be empty and unalterable. The following is the basic `TestBase` definition:

```
TestBase {
  Parameters {}
  FlowVariables {}
  PreActions {}
  TestExec;
  PostActions {}
  PassActions {}
  FailActions {}
}
```

This is its shorthand equivalent:

```
TestBase {}
```

Subclause 35.3.3 shows the default `TestBase` definition and explains the constraints under which the user may alter it.

[46] The immediate base type may be specified by the `Inherit` statement. See 28.2 and 31.1.

27.3 Parameter initialization and assignment

When a test-type or flow-type is defined, each of its parameters is assigned a default value. Each parameter that does not have attribute `Optional`, shall have its default value overridden when that test-type or flow-type is instantiated. Initialization and assignment syntax for parameters is the same as for variables with one exception: parameters have an additional directionality qualifier, either `In`, `Out`, or `InOut`, indicating input, output, or both.

A mathematical expression shall be evaluated at each point of use during execution if necessary, e.g., when relational operators evaluate their arguments, when tester hardware registers are loaded, or when a mathematical expression containing a mutable variable is used as or assigned to a constant.

The example of Figure 70 shows a `Parameters` block as it appears in the definition of a test-type. It sets default parameter values however the rules demonstrated in this example also apply when initializing parameters during test and flow instantiation. The `PreActions` block demonstrates assignment rules.

```
 1 TestType ExParams {
 2   Parameters {
 3     In          Integer i0 = 0;
 4     In    Const Integer i1 = 1;
 5
 6     // i2-i5: references to i0. Can reassign to other references
 7     // If Const, cannot change value by writing through reference
 8     InOut       Integer i2 = &i0; // Can change value
 9     InOut Const Integer i3 = &i0; // Cannot change value via i3
10     InOut       Integer i4 = &i0; // Same as line 8
11     InOut Const Integer i5 = &i0; // Same as line 9
12
13     // m0 is Const - cannot update spec var Meas field via m0
14     InOut Const Seconds m0 = &spec.cat.var.Meas;
15
16     // m1 is not Const - can update spec var Meas field via m1
17     InOut       Seconds m1 = &spec.cat.var.Meas;
18
19     // t0 is const (both from use of Const, and/or
20     // to use of non-Meas field of a spec variable).
21     InOut Const Seconds t0 = &spec.cat.var.Typ;
22
23     Out         Volts   v0 =  0V;
24   }
25   PreActions {
26     // Reassign i2 from i0 to i4. i4 also assigned to i0,
27     // so in effect, no change.
28     // Changing i2's value changes i4's (and i0's) value
29     i2 = &i4;
30
31     // i3 is now reference to i1. Since i3 is Const Integer,
32     // cannot change value of i1 through i3
33     i3 = &i1;
34
35     // Update values through references
36     i4 = i1; // set value of i4's ref (i0) to i1 value. i0 = 1
37     i2 = 2;  // set value of i2's ref to 2. i0 = 2
38   }
39 }
```

Figure 70—Example: parameter initialization

At the user's discretion, individual array dimensions may be specified or not, e.g.:

```
1 TestType Example
2 {
3   Inherit TestBase;
4   Parameters {
5     InOut BinSpec passBin { Required; BinType Axis; }
6     InOut Const Limits  limsarray[passBin.size()] { Units "s"; }
7     InOut Integer a[2][];
8   }
9 }
```

The following line comments refer to the TestType example above:

- Line 5: Overrides `TestBase` attributes on inheritance to make `passBin` a required variable that shall be set to a `Pass` axis, `BinType` attribute `Pass` having been set in `TestBase`.
- Line 6: `limsarray` shall refer to a one dimensional array of the same size as the `passBin` axis. Its contents shall not be altered by this test.
- Line 7: array `a` is required to refer to a two dimensional array whose first dimension is of size 2 and whose second dimension may be of any size. Identifier `a` shall reference the same object for the life of the instantiation of `Example`.

Once a non-constant array reference parameter is initialized, it may be assigned another array of the same size and units.[47] Alternatively, an array parameter may be assigned a scalar of the same units which sets every member of the referenced array to the same value. An individual element of the array may be singled out via one or more bracket enclosed indices, each a scalar integer value zero or greater, and assigned a scalar of the same units changing the value of that array element.

27.4 Parameter types

Parameter types represent data structures. Specific types may represent a further specialization of one of these classifications. All parameter types have identifiers, constraints, attributes, functions, and operators.

All `FlowVariables` types shown in Table 11 (in 17.5) may be used as `Parameter` types. When used as parameters, `FlowVariables` types shall be designated as `In`, `InOut`, or `Out`.

Additional parameter types representing STIL block types, as shown in Table 19 are also permitted.

[47] The reference parameter points to the new array, i.e., not an array copy operation.

Table 19—STIL block parameter types

Type name	IEEE Std	User-settable constraints	Type-specific functions, operators, and selectors
`BinSpec`	STIL.4	BinType (<Pass|Fail>)(,<Group|Axis|Bin>);	
`Category`	STIL.0	Type (<Timing|DCLevels>)	**size(), []**
`DCLevels`	STIL.2		
`DCSequence`	STIL.2		
`DCSets`	STIL.2		
`PatternBurst`	STIL.0		
`PatternExec`	STIL.0		
`Selector`	STIL.0	Type (<Timing|DCLevels>)	
`Signal`	STIL.4		
`SignalGroup`	STIL.4	IOType (<In|InOut|Out>) StaticType (<Supply|Ground|Open|Level>) SigType (<Analog|Digital|AnalogDigital>)	**size(), at(*integer*)**
`Spec`	STIL.0	Type (<Timing|DCLevels>)	**size(), []**
`SpecVariable`	STIL.4	***Units*** *"*units_expr*";* constant/mutable Meas field	units(), meas() or Meas, min() or Min, typ() or Typ, max() or Max
`TestType`	STIL.4		
`Timing`	STIL.0		

Detailed parameter type descriptions follow. Parameter definition examples are as they would appear inside a test-type or flow-type `Parameters` block. Function, operator, and selector usage examples are shown as they would appear in a test-type or flow-type actions block or a test or flow instantiation block.[48]

BinSpec: used to pass a reference to a structure defined in the `SoftBinDefs` block which is referred to in the `BinMap` block which is referred to in the `TestProgram` block. Except for the setting and clearing of bins and the associated side effects on counters, `BinSpec` attributes cannot be changed at runtime. Attribute keyword `BinType` constrains the type of binning structure which is legal for that parameter. Qualifiers `Pass` or `Fail` constrain the parameter to a `Pass` or `Fail` group, axis, or bin, respectively. Qualifiers `Group`, `Axis`, or `Bin` constrain the parameter to a group, axis, or bin respectively. These two groups of qualifiers can be used in conjunction to further constrain a legal parameter to a pass bin, for example. Bin actions are applicable to a `BinSpec` parameter and are covered in 22.3, 22.5, and Clause 33. Alternate `BinSpec` parameter definition examples follow:

```
1     InOut BinSpec failBin = None { BinType Fail; }
2     InOut BinSpec failBin = None { BinType Fail, Group; }
3     InOut BinSpec passBin = None { BinType Pass, Axis; }
4     InOut BinSpec failBin = None { BinType Fail, Bin; }
5     InOut BinSpec failBin = None;
```

Category: used to pass a reference to a STIL.0 `Category` structure. With the exception of the spec variable `Meas` field, no STIL.4 code shall change the expressions inside a `Category` block. `Parameter` qualifier `Const` shall prevent assignment to the spec variable `Meas` field. Function `size()` returns the number of variables defined in the category block. Operator `[]` accesses the Nth variable in that block where numbering begins at zero. A definition example:

```
InOut Category cat = None;
```

[48] The difference between an action and an instantiation parameter assignment statement is that during instantiation, a value may be assigned to a constant parameter.

Usage is specified on a per test or flow basis but is generally expected to be commensurate with STIL.0 `PatternExec`. Function and operator usage examples:

```
1     catsz = cat.size();         // # variables in Category cat
2     specvar = &cat[0];           // First variable in Category cat
3     specvar = &cat[catsz - 1];  // last variable in Category cat
```

DCLevels: used to pass a STIL.2 `DCLevels` structure. No STIL.4 syntax shall change the expressions inside a `DCLevels` block; however, non-STIL.4 code may. The definition form is consistent with `Category` above. Usage is specified on a per test basis but is generally expected to be commensurate with STIL.2 `PatternExec`.

DCSequence: shall be used to pass user-defined sequences only, i.e., it shall be illegal to pass predefined sequences `InitialSetup`, `PowerRaise`, `PowerLower`, and `EndOfProgram`. No STIL.4 syntax shall change the expressions inside a DCSequence block.

NOTE—STIL.4 currently has no way to execute a user-defined `DCSequence`; therefore, a non-STIL.4 library test is required to execute it.

DCSets: used to pass a STIL.2 `DCSets` structure. No STIL.4 syntax shall change DCSets. The definition form is consistent with `Category` above. Usage is specified on a per test basis but is generally expected to be commensurate with STIL.2 `PatternExec` usage.

PatternBurst: used to pass a reference to a STIL.0 `PatternBurst` structure. A test's `TestExec` that consists of non-STIL.4 code, may temporarily override start and/or stop locations as per `VecLocation` parameters. The definition form is consistent with `Category` above. `PatternBurst` usage is specified on a per test basis but is generally expected to be commensurate with STIL.0 `PatternExec`.

PatternExec: used to pass a reference to a STIL.0 `PatternExec` structure. No STIL.4 syntax changes `PatternExec` data (see `PatternBurst`). The definition form is consistent with `Category` above. Usage is specified on a per test basis.

Selector: used to pass a STIL.0 `Selector` structure. No STIL.4 syntax shall change `Selector` data. This structure shall be used to select which of the four spec variable values (`Meas`, `Min`, `Typ`, or `Max`) to use for each spec variable (may be applied to `SpecVariable`, `DCLevels`, and/or `Timing`). The definition form is consistent with `Category` above. Usage is specified on a per test basis but is generally expected to be commensurate with STIL.0 `PatternExec`.

SignalGroup: used to pass a *sigref_expr* or `None` if optional. Signal and signal-group names referred to by the *sigref_expr* if any shall be in scope. To be in scope, a signal or signal-group names shall be defined in the unnamed `Signals` or `SignalGroups` blocks respectively, or brought into scope via the `Device` block.[49] Parameter constraints may optionally be set to `SigType` and either `IOType` or `StaticType`. When constraints are absent, any signal or signal group may be passed. Constraints shall apply to each individual signal, e.g., passing a group that contains signals of type `In` and some of type `Out` does not meet a constraint that calls for type `InOut`. Here is more detailed description of constraints:

[49] When using `SignalMap`, only the top-level unnamed `Signals` and `SignalGroups` blocks are in scope. When using the `Device` block, the top-level unnamed `Signals` and `SignalGroups` blocks plus `Signals` and `SignalGroups` blocks referenced by the `Device` and `Chip` blocks are in scope.

a) `IOType`:
`In`: `SignalGroup` containing only `In` and `InOut` signals shall be legal.
`Out`: `SignalGroup` containing `Out` and only `InOut` signals shall be legal.
`InOut`: `SignalGroup` containing only `InOut` signals shall be legal.

b) `StaticType`:
`Supply`: `SignalGroup` containing only `Supply` signals shall be legal.
`Ground`: `SignalGroup` containing only `Ground` signals shall be legal.
`Open`: `SignalGroup` containing only `Open`, e.g., unused package pins, signals shall be legal.
`Level`: `SignalGroup` containing only `Level`, e.g., low current reference voltage, signals shall be legal.

c) `SigType`:
`Analog`: `SignalGroup` containing only `Analog` or `AnalogDigital` signals shall be legal.
`Digital`: `SignalGroup` containing only `Digital` or `AnalogDigital` signals shall be legal.
`AnalogDigital`: `SignalGroup` containing only `AnalogDigital` signals shall be legal. These signals may switch dynamically between being analog or digital.

The general form allows for omission of the `Const` modifier which applies to `SignalGroup`:

```
1     InOut Const SignalGroup siggrp { IOType In; }
2     InOut Const SignalGroup siggrp { SigType Digital; IOType In; }
3     InOut Const SignalGroup siggrp { StaticType Supply; }
4     InOut Const SignalGroup siggrp { SigType Analog; StaticType Level; }
5     InOut Const SignalGroup siggrp { StaticType Open; }
```

Note that all `siggrp` alternative parameter definitions are initialized to `None` by default.

`SignalGroup` supports function `size()`, and `at(`*integer*`)`. Function `size()` returns the number of individual signals in the group. The `at` function indexes the group and returns an individual signal. The first signal is `siggrp.at(0)`, the last is `siggrp.at(siggrp.size() - 1)`. Negative indices and indices greater than group size – 1 shall be illegal; therefore, indexing an empty signal-group, a group of size 0, shall be illegal. The expression `siggrp.at(0).name()`, for example, would return the name of the first signal as a string if `siggrp` was not empty and had a value other than `None`.

Spec: used to pass a a reference to a STIL.0 `Spec` structure. With the exception of spec variable `Meas` fields, no STIL.4 code shall change the definitions inside a `Spec` block. Parameter qualifier `Const` shall prevent assignment to the `Meas` field of any variable contained herein. Function `size()` returns the number of categories defined in the `Spec` block. Operator `[]` accesses the Nth `Category` in that block. Numbering begins at zero. A definition example:

```
InOut Spec spec = None;
```

Usage is specified on a per test basis but is generally expected to be commensurate with STIL.0 `PatternExec`. Function and operator usage takes the same form as `Category`.

SpecVariable: used to pass a reference to a STIL.0 spec variable structure. The spec variable has `Meas`, `Min`, `Typ`, and `Max` fields and is defined in a `Spec` block or in a `Category` block which is defined in a `Spec` block. The `Min`, `Typ`, and `Max` fields of a spec variable are immutable by definition, i.e., their mathematical expressions shall not be altered. Their actual values, however, may change if the mathematical expressions depend directly or indirectly on a `Meas` field (see `Selector`). In the following definition example, only the `Meas` field is affected by the `Const` qualifier:

```
InOut Const SpecVariable specvar = None;
```

Without the `Const` qualifier, `specvar.Meas` may be set by the test, with the `Const` qualifier, it shall not.

Usage examples:

```
1     specvar = spec.Categories[I].Variables[J];
2     specvar = &spec.cat.var;
3     specvar.Meas = &testresult;
```

The following line comments apply to the specvar example above:

- Line 1: a test instantiation statement initializing `specvar` to the `J`th spec variable in the `I`th `Category` under `Spec spec`, where I and J are of type `Integer` with values of zero or greater.
- Line 2: a test instantiation statement initializing `specvar` to spec variable `var` in `Category cat` under `Spec spec`.
- Line 3: a test-type action statement assigning the value of `testresult` to the `Meas` field of `specvar`. This is illegal in this example but would be legal were parameter `specvar` defined without the `Const` qualifier. Assuming it is legal, i.e., `testresult` units shall also match `specvar` units, any subsequent access to the spec variable from this or any other test, via expression `spec.cat.var` for example, shall reflect the new `Meas` field value.

TestType: used to pass a reference to a STIL.4 test, i.e., an instantiation of *TestType* or any type derived from *TestType*. If for example, `TestBase` is specified as the *TestType*, this is no constraint at all since all test-types are derived from `TestBase`. If `StdFunctional` is specified as the *TestType*, then only an instantiation of `StdFunctional` or a type derived from it may be passed in. One application is for defining a generic histogram generating test-type that loops over the user's chosen test. Applying parameter qualifier `Const` prevents the assignment of values to parameters by, e.g., the histogram generating test.

Timing: used to pass a STIL.0 `Timing` structure. No STIL.4 syntax changes the expressions contained in the `Timing` block; however, non-STIL.4 code may. The definition form is consistent with `Category` above. Usage is specified on a per test basis but is generally expected to be commensurate with STIL.0 `PatternExec`.

Constraints are either intrinsic or user-settable. Intrinsic constraints are implied by the parameter type, e.g., type `Seconds` can only be initialized to or assigned a value of type `Seconds` whereas type `General` may be initialized to any numeric value. User settable constraints may occur in several locations. Type-modifier keyword `Const` occurs before the type-name and constrains a parameter to be immutable. Other constraints, located between braces along with attributes, restrict legal assignments to a subset of what might otherwise be legal.

All parameter types support functions `type()`, `name()`, and `description()`, which return type `String`, as shown in Table 14.

Parameters are defined in blocks preceded by keyword `Parameters`, which occurs inside test-type definitions only. For example, see Figure 71.

```
1 FlowVariables pgm {   // Top level definition of variable
2     Seconds prd = 10ns; // In TestType X, local prd hides this one
3 }
4 TestType X {
5     Parameters {      // Local definition of parameter
6         In Const Seconds prd { Optional; }
7     }
8 }
```

Figure 71—Example: global, top-level, and local FlowVariables

During initialization, the right-hand side, i.e., the value, of a variable or parameter to be assigned may be a reference to another that has already been initialized.

27.5 Parameter attributes

Parameter attributes appear following a definition statement, enclosed in braces. A parameter attribute is specified by its name followed by zero or more context specific arguments and terminated by a semicolon. For example:

```
1 FlowVariables {       // Global variables
2     Volts vdd = 3.3V
3         { Permissions RhsReadWrite; Description "Device power"; }
4 }
```

An attribute, if not explicitly specified, takes on its default value. Some attributes enable a GUI to impose constraints. When editing STIL.4 code via a text editor, constraint imposing attributes merely serve as directives which users may ignore at their own peril.

Table 20—Parameter attributes

Attribute	Argument	Default	Purpose
Description	*string*	Empty string	Declare intended use.
Optional | **Required**		**Required**	Optional marks a parameter that does not require an initialization value on test or flow instantiation
ReInitAt	ASYNC_EVENT_NAME	START	Describe which asynchronous event reinitializes this variable before executing its associated test. All variables shall be initialized on LOAD.
Units	*string*	units unspecified	To constrain units on type General only

27.6 Parameter operators and member functions

See Table 13, Table 14, and 17.7. All operators and member functions for FlowVariables also apply to Parameters.

For parameter type "InOut", an assignment statement in which the right-hand side (RHS) consists of a single object name prefixed by the '&' operator, the parameter on the left-hand side (LHS) is re-referenced to the object on the RHS. When the RHS consists of an expression other than that described above, the assignment is made to the object referenced by the LHS.

`BinSpec` may represent a `Pass` or `Fail` bin group, a `Bin`, or a `BinAxis`; for related functions, see 22.3 and 22.5.

27.7 Parameter array operations

See 17.8. All operations that apply to FlowVariable arrays also apply to Parameter arrays.

27.8 Spec variable access

A spec variable or its fields shall be accessed relative to its hierarchical location, e.g., `spec.cat.var` or `spec.cat.var.Meas`, assuming `spec` and `cat` are the names of a previously defined `Spec` and subordinate `Category` block containing variable `var`. STIL.0 usage of quoted variable names and period as a string concatenation operator may appear ambiguous, e.g., `"spec"."cat"."var"` may represent concatenated string `"speccatvar"` or a reference to a spec variable. STIL.4 code shall use only operator + for string concatenation.

Only the `Meas` field of a spec variable may be assigned a value, e.g.:

```
specname.catname.varname.Meas = 10ns;
```

When referring to a STIL.0 spec variable in any other STIL.4 expression, one of fields `Meas`, `Min`, `Typ`, or `Max` shall be selected, either directly, e.g., `specname.catname.varname.Typ`, or indirectly via an accompanying STIL.0 `Selector`.

- Each `Spec` block shall have a domain name, and each `Spec` block name shall be unique across all `Spec` blocks.
- A `Spec` block may be referred to directly by a parameter of type `Spec`. A `Category` block may be referred to directly by a parameter of type `Category`. A spec-variable may be referred to directly by a parameter of type `SpecVariable`. A spec-variable field may be referred to directly using a parameter or flow-variable of the appropriate type, e.g., Seconds, Volts, etc
- The reference hierarchy shall be constrained to one order: `Spec`, `Category`, variable, `Meas`|`Min`|`Typ`|`Max` field.
- The information contained in `Spec` blocks is not globally accessible to STIL.4. The spec block name is a significant part of the reference hierarchy. For example, `catname.varname.Typ` is not a legal STIL.4 reference whereas `specname.catname.varname.Typ` is (`catname` is not found in the top-level namespace, `specname` is).

SPEC_BLOCK_NAME.**size()** returns the number of categories in the spec block (type `Integer`). If the `Category` block is omitted in the `Spec` block specification, size() returns 1, i.e., there is one anonymous `Category` block.

SPEC_BLOCK_NAME.**name()** returns the name of the spec block as type `String`. When SPEC_BLOCK_NAME is a parameter, `name` returns the name of the original `Spec` block, not the name of the parameter.

We may iterate over categories and variables via indices. The first category or variables is accessed via index 0. In the following syntax, an explicit category name and variable name can be substituted for **Categories**[I] and **Variable**[J]. Category and variable name order is as specified in the original spec block.

The Spec block (**Spec** (SPEC_NAME) { . . . }) is now part of the spec.category.variable.selector hierarchy.

SPEC_BLOCK_NAME.**Categories**[I] returns the Ith category in a spec block, which can then be further indexed.

SPEC_BLOCK_NAME.**Categories**[I].**name()** returns a string containing the category name corresponding to the Ith category in a spec block

SPEC_BLOCK_NAME.**Categories**[I].**size()** returns the number of variables in the Ith category in a spec block

SPEC_BLOCK_NAME.**Categories**[I].**Variables**[J] returns the Jth variable in the Ith category within a spec block, suitable for further indexing. The value of the variable is determined by the currently-active selector, unless that selection is explicitly overridden by specifying `Min`, `Typ`, `Max`, or `Meas`.

SPEC_BLOCK_NAME.**Categories**[I].**Variables**[J].**name()** returns a string containing the variable name of the Jth variable in the Ith category within a spec block.

SPEC_BLOCK_NAME.**Categories**[I].**Variables**[J].[<**Min** | **Typ** | **Max** | **Meas**>] returns the Jth variable in the Ith category within a spec block. The value of the variable is determined by the specification of `Min`, `Typ`, `Max`, or `Meas`.

Data-type `SpecVariable` shall refer to a single `Spec` variable, e.g., as expressed by the hierarchy SPEC_BLOCK_NAME.CATEGORY_NAME.SPEC_VARIABLE_NAME. `SpecVariable` subcomponents, i.e., `Min`, `Typ`, `Max`, and `Meas`, can be accessed by appending the desired selector field to the `SpecVariable`. In addition, there are two other operators available for a `SpecVariable` variable: `.name()` which returns the `SpecVariable` name as a string, and `.units()` which returns the units of the `SpecVariable` as a string. Function `.name()` can be used to retrieve the spec-variable name as defined in a `Spec` block inside a `Test` or `TestFlow` via its `SpecVariable` parameter. Function `.units()` can be used to type-check the spec-variable passed into a `Test` or `TestFlow` to insure it has the expected units.

Semantic rules:

- The combination of *spec block name*+*category name*+*spec variable name* shall be unique. This is an extension of the STIL.0 rule stating that the combination of *category name*+*spec variable name* shall be unique.
- Zero or more of variable fields `Min`, `Typ`, `Max`, or `Meas` may be defined. Any field omitted in the spec block definition shall be initialized to value `None`.
- All spec-variable field units shall be the same, e.g., if one or more fields specify volts then any uninitialized fields, i.e., fields that are set to `'None'`, shall be deemed to have the same units.
- Units or the lack thereof may be specified for a spec-variable, e.g., *spec_expr* `'None'` is indeterminate, `'3'` specifies no units, and either `'3V'` or `'NoneV'` specifies volts. Once a spec-variable field is initialized to a value other than `'None'` or the `Meas` field is assigned a value (`None` is not a legal assignment value), the spec-variable shall remain constrained to the units of that value for the remainder of its existence. Keyword `Units` may be used to specify units when only the Meas field is intended to be used, i.e., all fields are set to None without units.
- A mutable `SpecVariable`'s `Meas` field may serve as the target of an assignment. A `SpecVariable`'s `meas()` function, although it returns the `Meas` value, shall not be the legal target of an assignment:

28. TestType definition (FlowExtended)

28.1 General

For each IEEE standard-defined test-type, its function and how it uses its parameters to perform that function should be described. See Clause 35 for standard definitions. The `TestExec` statement followed by a semicolon represents inherited or non-STIL.4 code. Alternatively, the `TestExec` statement may be followed by a brace enclosed block of STIL.4 flow-node statements invoking other standard tests. Standard documents may choose to use flow-node statements to unambiguously describe a specific standard test-types' behavior, whereas the implementation may be via non-STIL.4 code or flow-node statements, at the provider's discretion. Local standards, i.e., ATE vendor or organization-wide provided test-type libraries, should follow suit.

For test-types in general, the TestExec statement followed by a semicolon[50] executes the base type's TestExec or non-STIL.4 code. The `TestExec` statement followed by a brace enclosed block of flow-node statements overrides the base type's `TestExec`. Under the control of a UI, the block is intended to be seen as a black box in a production environment and in full detail in a development environment.[51]

28.2 TestType syntax

The general form of a `TestType` definition is as follows:

test_typedef_stmt ::=
TestType TEST_TYPE_NAME {
(< **Inherit** < **TestBase** | TEST_TYPE_NAME | FLOW_TYPE_NAME > ; |
Inherit < **TestBase** | TEST_TYPE_NAME | FLOW_TYPE_NAME > {
param_override_stmts
} >)
(**Parameters** {
((<**In** | **Out** | **InOut**>) *param_elements_stmt*)*
})
(**FlowVariables** { (*var_elements_stmt*)* })
(**PreActions** { (*action_stmt*)* })
(< **TestExec**; | (*flownode_stmt*)* | **TestExec** {(*flownode_stmt*)*} >)
(**PostActions** { (*action_stmt*)* })
(**PassActions** { (*action_stmt*)* })
(**FailActions** { (*action_stmt*)* })
}

For `In` and `Out` Parameters, the following definitions of *param_override_stmts* and *param_val_override_stmts* apply:

param_override_stmts ::=
(< PARAM_NAME { *param_attributes* } |
PARAM_NAME = *param_value_expr*; |
PARAM_NAME = *param_value_expr* { *param_attributes* } |
PARAM_NAME[] = { *param_attributes* }
PARAM_NAME[] = *value_expr*; |
PARAM_NAME[] = *value_expr* { *param_attributes* } |

[50] The absence of the `TestExec` statement is equivalent to the `TestExec` statement followed by a semicolon.
[51] A flow differs from a test in that its `TestExec` details are intended to be visible in both production and development environments.

PARAM_NAME[] = ([*value_expr* (,*value_expr*)+])+; |
PARAM_NAME[] = ([*value_expr* (,*value_expr*)+])+ { *param_attributes* }
>)*

param_val_override_stmts ::=
(<PARAM_NAME = **None(**units**)***;* |
PARAM_NAME = *param_value_expr*; |
PARAM_NAME[] (= **None**(*units*)); |
PARAM_NAME[] = *value_expr*;
PARAM_NAME[] = ([*value_expr* (,*value_expr*)+])+;
>)*

For `InOut` Parameters, the following definitions of *param_override_stmts* and *param_val_override_stmts* apply:

param_override_stmts ::=
(<PARAM_NAME = { *param_attributes* } |
PARAM_NAME = &*stil_object_expr*; |
PARAM_NAME = &*stil_object_expr* { *param_attributes* } |
>)*

param_val_override_stmts ::=
(<PARAM_NAME (= **None**(*units*)); |
PARAM_NAME = &*stil_object_expr*;
>)*

The *param_override_stmts* are used to override the initialization and/or attributes of parameters defined in the inheritance chain during definition of derived types. Restrictions on how parameter attributes can be overridden are found in 26.3. New parameters created for this `TestType` are defined and initialized in the `Parameters` block. The definition and semantics of Inherit and *param_override_stmts* are in 28.2. The definition of the remainder of TestType elements (*param_elements*, *var_elements*, *action_stmt*, and *flownode_stmt*) are in 27.1, 17.2, and 30.2, respectively.

param_val_override_stmts are used to override parameter initialization values during instantiation.

Meta-type *param_value_expr* shall be of the same type as the type of param `PARAM_NAME`, or be convertible to that type. *param_value_expr* is defined in 27.1.

The definition of `TestType` block elements other than `Inherit` is in Clause 27. Any action block specified in this context shall completely override the inherited block of the same name in the base test-type.

Inherit: the optional inherit statement denotes the base type from which `Parameters`, `PreActions`, `TestExec`, `PostActions`, `PassActions`, and `FailActions` are inherited. The absence of an inherit statement is equivalent to `Inherit TestBase`. Base type parameter default initialization values may be overridden within the curly braces following the base type name.

TestExec: followed by a semi-colon, a reference to the base type's `TestExec`[52] or a standard or local library test-type's non-STIL.4 code.

(*flownode_stmt*)*: this form traverses flow-nodes beginning with the first and executes the test or flow associated with each. If there are no flow-node statements, the behavior is the same as the `TestExec` form

[52] The immediate base type may be specified by the `Inherit` statement. See 28.2 and 31.1.

above. If there are one or more flow-node statements, the behavior is the same as the `TestExec` form below. The definition of *flownode_stmt* is in Clause 30.

TestExec {(*flownode_stmt*)*}: this form shall override the base type's `TestExec`.

28.3 TestType example

The following is an example of test-type `TightAcFnc` which tests the device under test (DUT) at minimum and maximum timing edge values via the two selector parameters, sets bin `AcFnc` and either stops on fail, or continues on pass:

```
 1 TestType TightAcFnc {
 2     Parameters {
 3         InOut Selector     selmin;
 4         InOut Selector     selmax;
 5         InOut Timing       tim;
 6         InOut DCLevels     dclev;
 7         InOut PatternBurst patburst;
 8     }
 9     TestExec {
10         TestExec MyFunctional {
11             sel      = &selmin;
12             tim      = &Parent.tim;
13             dclev    = &Parent.dclev;
14             patburst = &Parent.patburst;
15         }
16         TestExec MyFunctional {
17             Local.sel      = &Parent.selmax;
18             Local.tim      = &Parent.tim;
19             Local.dclev    = &Parent.dclev;
20             Local.patburst = &Parent.patburst;
21         }
22     }
23 }
```

Figure 72—Example: TestType calling subflow using inline instantiation of other TestTypes and implicit standard FlowNode

Assumptions were made to keep the example compact. The following line comments refer to Figure 72:

- Lines 3–7: parameters are added to the local namespace; each is required and initialized to `None` by default.
- Lines 10–15: upon instantiation of `TightAcFnc`, an anonymous inline copy of `TestType MyFunctional` is created. Its default initialization values are overridden by the assignment statements. A copy of the standard flow-node is created and its `TestExec` is made to refer to the anonymous copy of `MyFunctional`.
- Lines 16–21: same comments as for lines 10–15.

29. Test

29.1 General

The Test statement contains a set of parameter assignments and references a TestMethod or TestType. In FlowExtended, the Test statement instantiates a `Test` from a `TestType`, either previously defined or defined inline. Using any of the forms, the `Test` statement assigns test specific parameter data for execution. To permit a test parameter to refer to an unnamed block upon instantiation, all unnamed top-level blocks shall implicitly be assigned the identifier `Unnamed`.[53]

In FlowExtended, the local scope of a Test block includes parameters and local variables. Each variable and parameter name shall be unique in the local scope. Variables and parameters declared in the local scope hide variables of the same name higher in the containment hierarchy and ultimately the global scope. The global scope includes `Spec` block names, variables declared at the top level in an unnamed `FlowVariables` block, or one or more named `FlowVariables` blocks that are referenced in the `TestProgram` block.[54]

To specify variables and/or parameters in scopes other than the local scope, the following rules apply:

- To access a top-level variable that is not hidden by a local parameter or variable, simply refer to it by name.
- To access a top-level variable that is hidden by a local parameter or variable, use the notation `Global.VAR_NAME` which corresponds to `::VAR_NAME` in C++.
- To access a local variable, simply refer to it by name, or use the notation `Local.VAR_NAME`. The `Local` notation allows the programmer to indicate that the local scope should be used, regardless of whether or not this variable hides another by the same name at a higher level. The use of `Local.VAR_NAME` is analogous to the "this" pointer (`this->VAR_NAME`) from C++.
- To access a parameter of the test or flow executed by the flow-node from within the flow-node, use the notation `CurrentExec.PARAM_NAME`.
- For a test or flow to access the containing tests' or flow's parameter, use `Parent.PARAM_NAME`.
- For objects contained in other objects the notation `OBJECT_NAME.SUBOBJECT_NAME` may be used provided the subobject is accessible. A spec variable or parameter may be accessed this way whereas test or flow variables are inaccessible.

29.2 Test syntax

A STIL.4 input stream may contain any number of `Test` instance statements using any of the following forms:

a) The following definition of *test_instance_stmt* applies when using `Flow 2017` (Clause 7):

test_instance_stmt ::=
< **Test** TEST_NAME {
TestMethod METHOD_NAME;
(**MethodParameters** {
(**In** *method_param_type* PARAM_NAME = *param_value_expr*;)*

[53] An optional test parameter is set to `None` by default. The right-hand side assignment value `Unnamed` is the only way to make the parameter reference an unnamed top-level block.
[54] Strictly speaking, only the unnamed variables block is truly global; however, named variables blocks referenced in the `TestProgram` block become global to the test program which ties together all objects used to test the device.

```
        (Out method_param_type PARAM_NAME -> object_expr; )*
    })
  }
 >
```

When using `Flow 2017`, *test_instance_stmt* creates a `Test` block specifying a `TestMethod` name and optional input parameters and output value(s), creating a connection between a Test block and a named `TestMethod` with optional passing of input parameter values to the TestMethod and outputs from the `TestMethod`. `TestMethods` are not defined within STIL.4; therefore, define before use does not apply to the test method name.

The semantics of `In` and `Out` parameters are as follows:

- `In`: An input object passed to the TestMethod. Changes internal to the TestMethod will not update the object's value outside test.
- `Out`: An output value (from an output parameter PARAM_NAME, as defined by the TestMethod) assigned to the object represented by *object_expr*.

b) The following definition of *test_instance_stmt* applies when using `FlowExtended 2017` (Clause 7):

```
test_instance_stmt ::=
  < Test TEST_TYPE  TEST_NAME; |
    Test TEST_TYPE  TEST_NAME { param_val_override_stmts } |
    Test TEST_NAME {
      TestMethod METHOD_NAME;
      ( MethodParameters {
        (In method_param_type PARAM_NAME = param_value_expr; )*
        (Out method_param_type PARAM_NAME -> object_expr; )*
      })
    }
  >
```

The definition for *param_val_override_stmts* is in 28.2. During Test instantiation, parameter attributes shall not be overridden, and `Out` parameter values shall not be overridden.

param_value_expr ::= *value_expr* | *stil_object_expr*

method_param_type and *stil_object_expr* are defined in 6.11.
value_expr is defined in 17.2.

object_expr ::= <anything that evaluates to a named object, such as a FlowVariable or the `.Meas` field of a spec variable>

When using `FlowExtended 2017`, the *test_instance_stmt* definition has three forms:

- The first form is used only if all required TestType parameters have default values other than `None`.
- The second form may be used to override some or all default test parameter values of the `TestType`. Parameters in *param_override_stmts* with attribute `Required` (rather than `Optional`) and a default value of `None`, shall be initialized via a *param_override_stmts* statement with a value other than `None`.
- For the first and second forms, the semantics of `In`, `InOut`, and `Out` are as defined in Table 18.

— The third form creates a `Test` block specifying a `TestMethod` name and optional input parameters and output value(s), as described above for Flow 2017.

29.3 Test example

The example in Figure 73 shows the form of a `Test` block without a `TestType`.

```
 1 FlowVariables { // Global variables
 2   Volts   vx1  = 30mV;
 3   Seconds tx1  = 660ps;
 4 }
 5
 6 Test DSPtest1 {
 7   // DSPMethod has two output params, named time_val and vil
 8   TestMethod DSPmethod;
 9   MethodParameters {
10     In sigref_expr grp = 'AnalogOut';
11     In Hertz sampleRate = 200kHz;
12     In Decibels snr        = 53dB;
13     In Percent thd = 0.2%;
14     Out Volts vil -> vx1;          // vil assigned to vx1
15     Out Seconds time_val -> tx1;   // time_val assigned to tx1
16   }
17 }
18
19 Test DSPtest2 {
20   // DSPMethod has two output params, named time_val and vil
21   TestMethod DSPmethod;
22   MethodParameters {
23     In sigref_expr grp  = 'AnalogOut';
24     In Hertz sampleRate = 96kHz;
25     In Decibels snr     = 98dB;
26     In Percent thd      = 0.01%;
27     Out Volts vil -> vx1;          // vil assigned to vx1
28     Out Seconds time_val -> tx1;   // time_val assigned to tx1
29   }
30 }
31 Flow main {
32   FlowNode {
33     TestExec DSPtest1;
34     ExitPorts {
35       Port tx1 > 30ns {} Return; // Conditionally exit flow
36       Port True {} Next;
37     }
38   }
39   FlowNode {
40     TestExec DSPtest2;
41     ExitPorts {
42       Port True {} Next;
43     }
44   }
45 }
```

```
46 TestProgram TEST_PROGRAM_NAME {
47   EntryPoints {
48      On START main;
49   }
50 }
```

Figure 73—Example: Test block without TestType

The example in Figure 74 shows a `Test` statement that instantiates from a defined `TestType`.

```
 1 // Test-types are most likely defined in a library
 2 TestType DSPtype {
 3   // TestType description: vil sets waveform base level for
 4   // waveform generator hardwired to DUT input.
 5   // TestExec raises waveform base level by 20uV increments until
 6   // input waveform amplitude becomes nonviable as judged by output
 7   // on parameter grp
 8   // Overridable PostActions restore last viable amplitude.
 9
10   // TestType parameter description:
11   //   grp: input waveform amplitude viability
12   //   sampleRate: waveform generator sample rate
13   //   snr: waveform generator signal-to-noise ratio
14   //   thd: waveform generator total harmonic distortion
15   //   vil: DUT input waveform average base level, see TestType desc
16   //   time_val: TestExec test time
17   Parameters {
18                                  // Constraints
19     InOut Const SignalGroup grp { IOType Out; SigType Analog; }
20     In    Const Hertz       sampleRate;
21     In    Const Decibels    snr;
22     In    Const Real        thd = 0.2/100 { Optional; } // 0.2%
23     InOut       Volts       vil;
24     Out         Seconds     time_val;
25   }
26   // TestExec changes parameter values as per TestType description
27
28   PostActions {
29     vil = vil - 20uV; // Sets vil to last viable amplitude
30   }
31   // Default PassActions are a no-op (would set pass-bin if
32   // one were passed in)
33   // Default FailActions propagate fail status up to entry-point
34 }
35
36 Signals {
37   AnalogOut Out+Analog; // Signal "AnalogOut" is an analog output
38 }
39
40 FlowVariables {                   // Global variables
41   Volts   vx1  = 30mV;
42   Seconds tx1  = 660ps;
43 }
44 Test DSPtype DSPtest1 {         // TestType DSPtype instantiation
45   grp        = &AnalogOut;
46   sampleRate = 200kHz;
47   snr        = 53dB;
```

```
48   // Param thd not specified - uses DSPtype default value (0.2%)
49   vil        = &vx1; // Binds parameter vil to global vx1
50 }
51
52 Test DSPtype DSPtest2 { // TestType DSPtype instantiation
53   grp        = &AnalogOut;
54   sampleRate = 96kHz;
55   snr        = 98dB;
56   thd        = 0.01/100; // 0.1%
57   vil        = &vx1;    // Parameter vil is reference to global vx1
58 }
59 Flow main {
60   FlowNode {
61     TestExec DSPtest1; // Behavior same as Figure 69 up to here
62     PostActions {
63       tx1 = CurrentExec.time_val;
64    }
65     ExitPorts {
66       Port tx1 > 30ns {} Return; // Conditionally exit flow
67       Port True {} Next;
68     }
69   FlowNode {
70     TestExec DSPtest2;
71     ExitPorts {
72       Port True {} Next;
73     }
74   }
75 }
76
77 TestProgram TEST_PROGRAM_NAME {
78   EntryPoints {
79      On START main;
80   }
81 }
```

Figure 74—Example: Test statement using defined TestType (FlowExtended)

The following comments compare the two `Test` examples:

- Both `Test` examples define and assign identical parameter data, and modify `FlowVariables` `vx1` and `tx1`.
- The first example does not use a `TestType`. Instead, it references a `TestMethod` named `DSPmethod` not defined in STIL.4.
- The second example defines a `TestType` named `DSPmethod` that can be inherited by other `TestTypes` and instantiated by `Test` statements.

30. FlowNode

30.1 General

A flow-node is a node in a directed graph representing a finite state machine. The flow-node has one entrance and in theory, any number of exit ports of which one shall be chosen at execution time.

30.2 FlowNode syntax

The *flownode_stmt*, referred to in other parts of this document takes the following form:

flownode_stmt ::=
< **FlowNode** (NODE_NAME) {
(**Position** *integer integer*;)
(**Disable** (*boolean_expr*);)
(**TestNumber** *unsigned_expr*;)
(**InheritParentCategorySelector** | // Use either Inherit… or Category/Selector. Cannot use both
(**Category** CATEGORY_NAME;)*
(**Selector** SELECTOR_NAME;)*)
(**PreActions** { (*action_stmt*)* })
< (**TestExec** *reference_stmt*)+ | **TestExec**; >
(**PostActions** { (*action_stmt*)* })
(**ExitPorts** {
((PORTLABEL:) **Port** *boolean_expr* { (*action_stmt*)* } *next_stmt*;)+
})
} |
TestExec *reference_stmt* >

NODE_NAME: The flow-node name serves as the target of a connection from another flow-node or itself via *next_stmt*. The standard flow-node is designed so that most of the time the name is unnecessary. The flow-node name shall be unique within the block that contains the flow-node.

action_stmt: Optional operation invoked before and/or after the `TestExec`. Table 21 shows action statements and where each is legal.

next_stmt: The *next_stmt* shall define the next step in the navigation of the execution flow if the condition of the Port *boolean_expr* is met. The "Port Actions" and "Branching Actions" columns in Table 21 show which action statements are legal.

Position: An optional statement with an x/y pair of unsigned integers to represent relative position of the `FlowNode` within its `Flow` for documentation or illustrative purposes. The unsigned integer values have no units or implied scale. The first number represents relative position on an x-axis with the value 0 representing the leftmost position. The second number represents relative position on a y-axis with the value 0 representing the topmost position. This statement has no impact upon flow navigation.

Disable: An optional keyword with optional *boolean_expr* used to skip the `FlowNode`. If keyword is present without expression, or if keyword is present with a true expression, flow navigation shall continue with the next `FlowNode`, or return if the last `FlowNode`.

TestNumber: An optional *unsigned_expr* assigned to the `FlowNode`. In `FlowExtended`, if a `Test` (rather than a `Flow`) is invoked by the `TestExec`, this `TestNumber` overrides the `testNumber` parameter in the referenced `Test`.

Category/Selector context shall be cleared at the beginning of the FlowNode. When Category and Selector are required to resolve spec references, they shall be set in either the FlowNode (as described below) or as a parameter to the test being executed.

It is recommended that Category/Selector context be set either in the FlowNode or in the Test or Flow, but not both.

InheritParentCategorySelector: Inherits the Category/Selector context from the FlowNode which called the subflow containing this flownode. If present, neither Category nor Selector shall be specified.

Category: An optional Category block name or expression evaluating to a Category block name. Selects a category, which defines the values of spec variables to be used for this FlowNode. This selection remains for the duration of the FlowNode, including all post-execution actions, unless overridden at the Test or Flow level, including overriding by the value `None`.

Selector: An optional Selector block name or expression evaluating to a Selector block name. Selects a Selector block, which defines the Min, Typ, Max, or Meas values to be used for each spec variable that is referenced by the Category selected for this FlowNode. This selection remains active for the duration of the FlowNode, including all post-execution actions, unless overridden at the Test or Flow level, including overriding by the value `None`.

PreActions: An optional list of actions performed prior to the `TestExec`. Action statements are described in Clause 33.

TestExec: When a single `TestExec` statement occurs within the `FlowNode` block, it invokes a `Test` or `Flow` using a *reference_stmt* as described in 26.4. `FlowExtended` allows for keyword `TestExec` followed immediately by a semicolon within the `FlowNode` block. This shall be legal only for the top-level standard flow-node definition.

Multiple `TestExec` statements may occur within the `FlowNode` block. Conceptually, this may be described as invoking a single `Flow` which implements wrapping `FlowNodes` for each `TestExec`. Only the `FlowExtended` syntax allows for test-type definitions.

When the `TestExec` statement occurs within a `Flow` block or test-type, the *reference_stmt* shall be executed and flow navigation shall by default proceed to the next *flownode_stmt*. In this case, there is no value for the `FlowNode` NODE_NAME, `TestNumber`, or `execResult`.

PostActions: An optional list of actions performed after the `TestExec`. Action statements are described in the Clause 33.

execResult: The `FlowNode` contains a read-only `execResult` member of type `ExecResult`. This member is set by the `TestExec`. The mechanics of setting execResult is not defined as part of STIL. The Boolean expression statements in the `ExitPorts` may make use of this member by testing its value against possible enumerators of `ExecResult`.

ExitPorts: A block containing a list of `Port` statements. Each `Port` statement includes a Boolean expression. These Boolean expressions are evaluated in the order that the ports appear. At runtime, the first port with a Boolean expression that evaluates to true is triggered.

Port: A `Port` includes an optional PORTLABEL, a *boolean_expr*, a block of zero or more action statements (metatype *action_stmt*), and a *next_stmt*. In the `ExitPorts` block, the first port in the list which contains a *boolean_expr* evaluating to `True` is triggered. Once triggered, the port performs the actions enclosed in braces, e.g., set a soft bin and stop the flow. Thereafter, the *next_stmt* shall pass control to another flow-node or the `PostActions` block of the enclosing flow. Exit-port actions are described in Clause 33. The optional PORTLABEL is arbitrarily named. This name cannot be referenced. Setting the last port's Boolean expression to `True` guarantees a valid exit for the flow-node.

Refer to 35.3.2 for the standard flow-node definition.

Unspecified clauses, such as pre-actions, exit-ports, etc., shall use standard flow-node definitions and behaviors.

30.3 FlowNode examples

```
FlowNode StuckAt1 {
  TestNumber 41;
  TestExec testStuckAt1;
  ExitPorts {
    Port 'execResult==Pass' { } Next;
    FAIL: Port 'True' { SetBinStop failSCAN; }
  }
}
```

Figure 75—Example: FlowNode with ExitPorts

```
// Example A:
FlowNode {
  TestExec test1;
  TestExec test2;
}

// Example B:
FlowNode {
  TestExec StdFlow {
    TestExec test1;
    TestExec test2;
  }
}

// Example C:
FlowNode {
  TestExec StdFlow {
    FlowNode { TestExec test1; }
    FlowNode { TestExec test2; }
  }
}

// Example D:
FlowNode {
  PreActions {}
  TestExec StdFlow {
    FlowNode {
      PreActions {}
      TestExec test1;
      PostActions {}
      ExitPorts {
        Port True {} Next;
      }
    }
    FlowNode {
      PreActions {}
      TestExec test2;
      PostActions {}
      ExitPorts {
        Port True {} Next;
      }
    }
  }
```

```
44   PostActions {}
45   ExitPorts {
46     Port True {} Next;
47   }
48 }
```

Figure 76—Example: equivalent FlowNode specification forms (FlowExtended)

Examples A, B, C, and D in Figure 76 are functionally equivalent. Every explicitly or implicitly created flow-node executes standard flow-node actions. When `FlowExtended` standard `FlowNode` and `TestBase` are in effect and `test1` passes, `test2` is executed, if `test1` fails, the inline-instantiated `StdFlow`'s `PostActions` are executed.

31. FlowType definition (FlowExtended)

31.1 FlowType syntax

The general form for the definition of a flow-type follows:

flow_typedef_stmt ::=
FlowType FLOW_TYPE_NAME {
(< **Inherit** < **TestBase** | FLOW_TYPE_NAME >; |
Inherit < **TestBase** | FLOW_TYPE_NAME > {
param_override_stmts
} >)
(**Parameters** {
((<**In** | **Out** | **InOut**>) *param_elements_stmt*)*
})
(**FlowVariables** { (*var_elements_stmt*)* })
(**PreActions** { (*action_stmt*)* })
(**TestExec**;)
(**PostActions** { (*action_stmt*)* })
(**PassActions** { (*action_stmt*)* })
(**FailActions** { (*action_stmt*)* })
}

The definition and semantics of `Inherit` and *param_override_stmts* are in 28.2. The definition of the remainder of the `FlowType` elements (*param_elements*, *var_elements*, and *action_stmt*) can be found in 27.1, 17.2, and 30.2 , respectively.

Behavior is identical to `TestType` with regard to type definition. On instantiation, however, a `FlowType TestExec` statement may be overridden whereas a `TestType TestExec` statement shall not.

32. Flow

32.1 General

A Flow statement is a container for zero or more FlowNodes.

In FlowExtended, the Flow statement the `Flow` statement instantiates a `Flow` from a `FlowType`, either previously defined or defined inline. Using any of the forms, the `Flow` statement assigns flow specific parameter data for execution. To permit a flow parameter to refer to an unnamed block upon instantiation, all unnamed top-level blocks shall implicitly be assigned the identifier `Unnamed`.[55]

In FlowExtended, the local scope of a Test block includes parameters and local variables. The rules for local vs. non-local scoping and means of accessing non-local variables are as described for Test parameters and variables in Clause 29.

32.2 Flow syntax

A STIL.4 input stream may contain any number of `Flow` instance statements using any of the following forms:

The following definition of *flow_instance_stmt* applies when using `Flow 2017` (Clause 7):

flow_instance_stmt ::=
    < **Flow** FLOW_NAME { (*flownode_stmt*)* }>

The following definition of *flow_instance_stmt* applies when using `FlowExtended 2017` (Clause 7):

flow_instance_stmt ::=
    < **Flow** FLOW_TYPE FLOW_NAME; |
      Flow FLOW_TYPE FLOW_NAME { *flow_init_stmts* }} |
      Flow FLOW_NAME { (*flownode_stmt*)* }
    >

flow_init_stmts ::=
    < *param_override_stmts*
      < **TestExec**; | (*flownode_stmt*)* | **TestExec { (***flownode_stmt***)* }** >
    >

The definition and semantics of *param_override_stmts* are in 28.2. Parameter override statements may be omitted if all required `FlowType` parameters have default values other than `None`. During Flow instantiation, parameter attributes shall not be overridden, and `Out` parameter values shall not be overridden.

The definition of *flownode_stmt* is in 30.2.

For the `FlowExtended 2017` definition, there are three forms of the `Flow` statement:

- The first `TestExec` form terminated by a semicolon shall execute the most immediate base type's `TestExec` in the inheritance hierarchy.
- The second form consisting of zero or more *flownode_stmt* may be used to override the flow-type's flow-node statements if any. If there is no *flownode_stmt*, the behavior is the same as the first form. The first flow-node is the flow entry point.
- The third form using keyword `TestExec` is identical to the second with one exception: if it is terminated by an empty pair of braces, the flow shall be empty, i.e., a no-op. In FlowExtended mode, the implied type shall be StdFlow.

[55] An optional test parameter is set to `None` by default. The right-hand side assignment value `Unnamed` is the only way to make the parameter reference an unnamed top-level block.

32.3 Flow examples

The example in Figure 77 shows a **Flow** using **Flow 2017** constructs. It creates two tests (whose content is not specified here), shows the use of those tests in a Flow (used as a subflow) which contains two FlowNodes with common usage of elements, and finally shows another flow which calls the subflow from one of its FlowNodes

```
 1 Test testStuckAt1 { }
 2 Test testStuckAt2 { }
 3 Flow scan {
 4   FlowNode StuckAt1 {
 5     TestNumber 41;
 6     TestExec testStuckAt1;
 7     ExitPorts {
 8       Port 'execResult==Pass' { SetBin bin1; } Next;
 9       Port 'True' { SetBinStop failSCAN; }
10     }
11   }
12   FlowNode StuckAt2 {
13     TestNumber 42;
14     TestExec testStuckAt1;
15     ExitPorts {
16       Port 'execResult==Pass' { } Next;
17       Port 'True' { SetBinStop failSCAN; }
18     }
19   }
20 }
21 Flow main {
22  FlowNode scanTests {
23    TestNumber 40;
24    TestExec scan; // Executes the subflow “scan” – defined above
25    ExitPorts {
26      Port 'True' { } Next;
27    }
28  }
29 }
```

Figure 77—Example: Flow in TestProgram block (using Flow 2017 constructs)

The example in Figure 78 shows a **Flow** using **FlowExtended 2017** constructs. It instantiates a flow inside the **TestProgram** block, and shows the use of both explicit FlowNodes, and implicit FlowNodes using the standard FlowNode (see 35.3.2)

```
 1 TestProgram basicflow {
 2   Test StdFunctional fncloose {
 3     failBin  = LooseFunct;
 4     patburst = burst1;
 5     tim      = acloose;
 6     dclev    = dcloose;
 7   }
 8   Test StdFunctional fnctight {
 9     failBin  = TightFunct;
10     passBin  = Passed;
11     patburst = burst1;
12     tim      = actight;
```

```
13     dclev    = dctight;
14   }
15   Flow StdFlow main {
16     FlowNode {           // Explicit default flow-node
17       TestExec fncloose;
18     }
19     TestExec fnctight;   // Implicit default flow-node
20   }
21   EntryPoints {
22     On START main;
23   }
24 }
```

Figure 78—Example: Flow in TestProgram block (using FlowExtended 2017 constructs)

Execution shall begin with an asynchronous event that causes an entry point to execute its test or flow which shall then execute its `PreActions`, `TestExec`, `PostActions`, and either `PassActions` or `FailActions`, in that order. If the `TestExec` block contains flow-nodes, explicitly or implicitly, the first flow node is executed. The flow node shall execute its `PreActions`, `TestExec`, `PostActions`, and the actions of the first `Port` in the `ExitPorts` block whose Boolean statement evaluates to `True`. Finally, the selected port specifies which flow node to execute next. Control may pass to flow-nodes contained in the same test or flow, to the `PostActions` of the containing test or flow, or in case of an exception, back to the originating entry-point.

33. Actions and flow control

Actions are commands executed from test, flow, or flow-node action blocks most of which directly or indirectly affect flow. The test and flow action blocks are named `PreActions`, `PostActions`, `PassActions`, and `FailActions`. Flow-node actions blocks are named `PreActions`, `PostActions`, and `Port`. Some actions are constrained to specific action block types, i.e., those belonging to a test, flow, or flow-node and either before or after `TestExec`. See Table 21.

Table 21—Actions and their legal locations

Action	Test/Flow	FlowNode		PreActions	PostActions	Pass/Fail Actions	Port Actions	Branching Actions
Bypass	✓	✓		✓				
ClearBin	✓	✓		✓	✓	✓	✓	
If/Else If/Else	✓	✓		✓	✓	✓	✓	
Next (NODE_NAME**)**		✓						✓
Return		✓						✓
SetBin *soft_bin_expr*	✓	✓		✓	✓	✓	✓	
SetBinStop *soft_bin_expr*	✓	✓			✓	✓	✓	
Stop	✓	✓			✓	✓	✓	
var_assignment_stmt	✓	✓		✓	✓	✓	✓	
ReExec	✓	✓			✓	✓	✓	
While	✓	✓		✓	✓	✓	✓	

Table 21 contains the list of available actions. The "Test/Flow" and/or "FlowNode" columns are checked if the corresponding action is legal in any part of those blocks. One or more of the remainder of the columns are checked if the corresponding action is legal within the specified `Test`, `Flow`, or `FlowNode` sub-block or, for the "Branching Actions" column, after the `Port` brace-enclosed actions block.

The syntax for use with binning actions allows for specifying single or multiple bins. It is described in 21.2. The following is a detailed description of all available actions, some accompanied by usage examples.

Bypass: Skip pre-actions after the `Bypass` statement and `TestExec`. Execution resumes at entry to the `PostActions` block.

ClearBin < BIN_VAR_NAME | *unary_ bin_expr* | *multi_bin_expr* >: applies to soft bins only and clears the bin(s) corresponding to the parameter. BIN_VAR_NAME refers to a variable of type `BinSpec`. See 21.2 for metaypes *unary_ bin_expr* and *multi_bin_expr*. With `SoftBinDefs` defined as per Figure 55 and Figure 57 and linked in via the `TestProgram` block `BinMap` statement, the following examples clear all soft bins in group `Fail` and axis `ClockSpeed`, respectively:

```
ClearBin Fail;
ClearBin Pass.BinAxes[ClockSpeed];
```

If (*boolean_expr* **)** ({ (*action*)* } | *action*)
(**Else If (** *boolean_expr* **)** ({ (*action*)* } | *action*))
(**Else** ({ (*action*)* } | *action*))

This statement has normal `if/else-if/else` semantics, i.e., one of the actions or action blocks is executed depending on the evaluation of the parentheses enclosed Boolean expression. For example:

```
If      (speed >= 3.00GHz)
  SetBin "3.00GHz";
Else If (speed >= 2.93GHz)
  SetBin "2.93GHz";
Else
  SetBin None;
```
[56]

Precondition: variable `speed` shall be defined in a top-level `FlowVariables` block and linked in via the current `TestProgram` block, or in one of the `Parameters` blocks in the inheritance chain, or in the local test-type's or flow-type's `FlowVariables` block. The bins named `"3.00GHz"` and `"2.93GHz"` shall be defined, uniquely named, and linked in via the `TestProgram` block.

Next (NODE_NAME): specifies which flow-node gains control after the current one. The `Next` statement without parameters passes control to the next flow-node statement in the user code.[57] For the last flow-node in a list only, keyword `Next` shall be equivalent to keyword `Return`. The optional flow-node name specifies the next flow-node to which control is to be passed.

ReExec: re-executes `TestExec`, then re-enters PostActions.

Return: stop execution of flow-nodes and perform post-actions of the test or flow that contains the flow-node with this instruction.

[56] Bin `None` usage requires `FlowExtended` mode.
[57] The standard `FlowNode`, which is invoked via syntactical shortcut `TestExec`, is also a target of statement `Next`.

SetBin < BIN_VAR_NAME | *unary_ bin_expr* | *multi_bin_expr* >: applies to soft bins only and sets the bin(s) corresponding to the parameter.[58] BIN_VAR_NAME refers to a variable of type `BinSpec`. See 21.2 for metatypes *unary_ bin_expr* and *multi_bin_expr*. With `SoftBinDefs` defined as per Figure 55 and Figure 57 and linked in via the `TestProgram` block `BinMap` statement, these examples all set the same soft bin:

```
SetBin Pass.BinAxes[ClockSpeed].Bins["2.93GHz"];
SetBin Pass.BinAxes[0].Bins[1]; // Via indices
SetBin Pass.Bins["2.93GHz"]     // Unambiguous specification
SetBin "2.93GHz";               // Unambiguous specification
SetBin 2;                       // Via bin number
```

SetBinStop < BIN_VAR_NAME | *unary_ bin_expr* | *multi_bin_expr* >: applies to soft bins only. BIN_VAR_NAME refers to parameter type `BinSpec`. Refer to the *unary_ bin_expr* and *multi_bin_expr* explanations. Statement `SetBinStop failBin` is semantically equivalent to the following:

```
1 SetBin failBin;
2 If (failBin != None)
3     Stop;
```

The following line comments refer to the SetBin example above:

- Line 1: `failBin` refers to parameter `BinSpec failBin`.
- Lines 2 and 3: the `Stop` statement is reached only if parameter `BinSpec failBin` is set to a user-defined bin, i.e., `None` is the default.

Stop: return to the caller, bubbling up through the levels of flow-nodes and tests, until the initiating entry-point is reached. Local code specified after the `Stop` statement at the current level is not executed. At each subsequent higher level, execute the `PostActions` and `PassActions` or `FailActions` for tests, but skip execution of `FlowNodes` and their `PostActions` and `ExitPort` actions.

var_assignment_stmt: assigns an expression to a variable. The following is a valid example assuming `varname` and `value` are defined in the current context:

```
varname = value;
```

While (*boolean_expr* **)** (*action* | { (*action*)* })

This statement loops over a single action or block of actions as long as the parentheses enclosed expression evaluates to `True`.

34. TestProgram

34.1 General

The `TestProgram` block assembles the elements required to define a test program. In FlowExtended, one or more `TestProgram` blocks may be defined in the input stream. The `Device` block covered in Clause 20 chooses which test program(s) to execute. Alternatively, on a tester running a single

[58] Bins are automatically cleared by the entry point `On START` event handler before its test is executed. Once a bin is set, it remains set until cleared explicitly by a `ClearBin` statement or implicitly by the next `On START` event.

TestProgram, the user may use SignalMap, Clause 19, to associate signals with pads, pins, and channels.

34.2 TestProgram syntax

The following definition of **TestProgram** applies when using Flow 2017 (Clause 7):

(**TestProgram** TEST_PROGRAM_NAME {
  (**FlowVariables** VAR_BLOCK_NAME;)*
  (**BinMap** BIN_MAP_NAME;)
  (**SignalMap** SIG_MAP_NAME;)
  entry_pts_stmt
})+

The following definition of **TestProgram** applies when using FlowExtended 2017 (Clause 7):

(**TestProgram** TEST_PROGRAM_NAME {
  (**FlowVariables** VAR_BLOCK_NAME;)*
  (**BinMap** BIN_MAP_NAME;)
  (**SignalMap** SIG_MAP_NAME;)
  (*typedef_stmt*)*
  (*instance_stmt*)*
  entry_pts_stmt
})+

typedef_stmt ::= < *test_typedef_stmt* | *flow_typedef_stmt* >

instance_stmt ::= < *test_instance_stmt* | *flow_instance_stmt* >

Multiple TestProgram blocks is a FlowExtended feature used in the Device block.

FlowVariables: this keyword introduces a reference to a FlowVariables block. The unnamed FlowVariables block is implicitly referenced, i.e., all its variable definitions are visible from within the TestProgram block. VAR_BLOCK_NAME is a reference to a FlowVariables block defined at the top level, i.e., the same level as TestProgram. In addition to variables in the top-level unnamed FlowVariables block, all variables defined in the referenced block(s) become visible within the scope of the TestProgram block. Since all TestProgram variables reside in the same namespace, if multiple FlowVariables blocks are referenced, each variable's name shall be unique across all blocks. A FlowVariables block is replicated with each reference, i.e., once for each instance of TestProgram.

BinMap: this keyword introduces a reference to a BinMap block. BIN_MAP_NAME is a reference to a BinMap block defined at the top level.

SignalMap: this keyword introduces a reference to a SignalMap block. It shall be used only when SignalMap is employed instead of Device. This is because a single TestProgram can choose which SignalMap to use but a Device block chooses which TestProgram to run. SIG_MAP_NAME is a reference to a SignalMap block defined at the top level. The unnamed top-level SignalMap block may specifically be referred to as SignalMap Unnamed.

typedef_stmt: this statement defines a test-type or flow-type that in the context of the TestProgram block is visible only within that block. A type definition local to the TestProgram block shall not have the same name as an existing top-level type definition. Test-type or flow-type definitions may also occur at the

top level however data brought into scope by `TestProgram FlowVariables`, `BinMap`, and `SignalMap` references are not visible to them. Top-level test-type or flow-type definitions are visible from inside every `TestProgram` block. See Clause 28 for *test_type_def_stmt* syntax and Clause 31 for *flow_typedef_stmt* syntax.

instance_stmt: this statement creates an instantiation of a test-type or flow-type. The instance is visible only from within the `TestProgram` block where this statement occurs and may be referenced by the *entry_pts_stmt*. An instance local to the `TestProgram` block of the same name as an existing top-level instance shall hide the top-level instance. Note that top-level instance statements shall not be permitted when the input stream contains more than one `TestProgram` block.

entry_pts_stmt: this statement associates one or more asynchronous events, each with a test or flow instance to be executed. See 34.4 for syntax.

34.3 TestProgram examples

The example in Figure 79 illustrates a typical usage when using Flow 2017 (Clause 7). It contains a SignalMap, a BinMap, and if the flows main and eowFlow rely on FlowVariables, the global unnamed FlowVariables block shall contain the needed FlowVariables.

```
1 TestProgram pgm {
2     SignalMap HandTestSigMap;
3     BinMap binmap;
4     EntryPoints {
5         On START main;
6         On WAFER_END eowFlow;
7     }
8 }
```

Figure 79—Example: TestProgram using Flow constructs

The example in Figure 80 illustrates a typical usage when using `FlowExtended 2017` (Clause 7).

```
1 TestProgram pgm {
2     FlowVariables globals;
3     BinMap binmap;      // Select bin map
4     Flow StdFlow main;  // Instantiate StdFlow as main
5     EntryPoints {
6         On LOAD None;
7         On START main;  // Execute Flow main when start is received
8     }
9 }
```

Figure 80—Example: TestProgram using FlowExtended constructs

For the sake of brevity, `FlowVariables` (Clause 17), `BinMap` (Clause 24), and `StdFlow` (35.7) definitions are not shown in Figure 80. Define before use rules apply. This `TestProgram` example is compatible with the `Device` block because it has no `SignalMap` reference. FlowVariables available to `pgm` include those in the named **FlowVariables** block `global`, and those in any unnamed **FlowVariables** block.

34.4 Entry points

Keyword **EntryPoints** occurs within the `TestProgram` block. It introduces a block of asynchronous events each of which triggers the execution of a test or flow using the following form:

entry_pts_stmt ::=
EntryPoints {
  (**On** ASYNC_EVENT_NAME *reference_stmt*)*
}

ASYNC_EVENT_NAME represents one of the enumerations of enumerated type `AsynchronousEvent` (see 35.1 for standard definitions). No event shall be listed more than once in the `EntryPoints` block. The definition of *reference_stmt* is in 26.4. See example in Figure 80.

The execution sequence emanating from entry-point `On START` triggers standard named DC-sequences (see keyword `DCSequence` in this document and STIL.2).

Semantics of fail status propagation to top include, for example, the following:

— The state of each entry-point shall be the state of the test or flow it points to.
— For the test or flow that an entry-point refers to, statements involving scope operator `Parent` are legal but shall be ignored.

34.5 Bin map

The `TestProgram` block `BinMap` statement specifies a named `BinMap` block. The following example shows the use of unnamed `SoftBinDefs` and `HardBinDefs` blocks. The `TestProgram` specifies a `BinMap` block, which uses those unnamed blocks and therefore does not need to specify either by name).

```
SoftBinDefs {
  Pass {
    Bin bin1 { Number 1; Color Green; }
  }
  Fail {
    Bin failSCAN { Number 10; Color Red; }
  }
}
HardBinDefs {
  Pass {
    Bin bin1 { Number 1; Color Green; }
  }
  Fail {
    Bin bin2 { Number 2; Color Red; }
  }
}
BinMap binmap {
  bin1 -> bin1;
  failSCAN -> bin2;
}
TestProgram pgm {
  BinMap binmap;
}
```

The next example shows the use of a named `SoftBinDefs` block.

```
SoftBinDefs softbindefs {
    Pass {
        Bin Passed;
    }
    Fail {
        Bin Functional;
    }
}

BinMap binmap {
  SoftBinDefs softbindefs;
}

TestProgram pgm {
    BinMap binmap;
}
```

This example uses `binmap` to make soft bin definitions accessible to the test program. For simplicity's sake the optional hard bin definitions and `binmap` soft to hard bin mappings are omitted. In FlowExtended mode, if there's no hard binning and the test program uses only bin `None`, neither bin definitions nor a bin map reference shall be required since bin `None` is defined by default.

35. Standard definitions

35.1 Standard enumerated types

35.1.1 General

In addition to allowing the user to define enumerated types, STIL.4 requires some standard enumerated types. Standard enumerated type definitions shall be accessed automatically by the STIL.4 compliant tool, i.e., they shall not require an `Include` statement.

A tool provider may extend the list of standard enumerations by adding to the end only. The expectation is that these and other standard types are provided as part of a tool library, i.e., the user should not have to provide the definitions as shown in 35.1.2 and 35.1.3. Standard enumerated types for FlowExtended (35.1.3) shall include those defined for Flow (35.1.2).

35.1.2 Standard enumerated types

```
Enum AsynchronousEvent {
    NEVER,         // event that never occurs - used to disable ReInitAt
    LOAD,          // Load executable test program
    LOT_START,     //
    WAFER_START,   //
    START,         // Start test or main flow
    TEST_ENTRY,    // Enter a test or flow
    RETEST,        // Retest DUT
    END,           // Stop test or main flow
```

```
    WAFER_END,     //
    LOT_END,       //
}
```

`AsynchronousEvent` enumerations are used with `EntryPoints`, reinitialization specification for variables via keyword `ReInitAt`, and functions such as `countSince`.

```
Enum ExecResult {
    NORESULT,       // No execution result
    PASS,           // Pass result
    FAIL,           // Fail result
    ERRORRESULT,    // Error result
}
```

`ExecResult` is used by keyword `execResult`.

35.1.3 Standard enumerated types (FlowExtended)

```
Enum BinGroup
{
    FAIL,
    PASS,
    NONE,
}
```

A `BinGroup` enumeration is returned by bin member function `getBinGroup()`.

```
Enum CheckResult {
    PASS,           //
    FAIL_UNITS,     // Units mismatch
    FAIL_BOTHLIM,   // Lower and upper limit
    FAIL_HILIM,     // Upper limit
    FAIL_LOLIM,     // Lower limit
    INDETERMINATE,  // Limits compared to result None
}
```

A `CheckResult` enumerations is returned by `Limits` member function `check`.

```
Enum FailMode {     // FlowExtended
    PASS,
    EXCEPTION_SOFT, // Software exception, e.g., divide by 0
    EXCEPTION_HARD, // Hardware exception
    FAIL_SETUP,     // Tester or device configuration
    FAIL_FNC,       // Functional or functional part of parametric
    FAIL_PRM,       // Parametric, e.g., search endpoints
    NOEXEC,         // Test's or flow's TestExec not executed
}
```

`FailMode` is used by `TestBase` parameter `failMode`.

```
Enum LevelGrp {
    BICMOS,
    BTL,
    CMOS,
    ECL,
```

```
    ETL,
    GTL,
    GTLP,
    I2L,
    LVCMOS,
    LVECL,
    LVPECL,
    LVTTL,
    NECL,
    PECL,
    TTL,
}
```

`LevelGrp` enumerations are used to infer input and/or output buffer voltage levels. STIL.4 does not specify the actual voltage levels associated with each enumeration. Current capabilities shall be specified separately on a per-buffer or buffer-type basis.

```
Enum LocType {
    LABEL,
}
```

`LocType` enumerations are used with type `VecLocation`.

```
Enum TestMode {
    PRODUCTION,     // Production
    ENGINEERING,    // Engineering/characterization
    DEVELOPMENT,    // Development/debug
}
```

`TestMode` enumerations are intended for flow control.

35.2 Standard global variables (FlowExtended)

Standard variables if used shall prepopulate the unnamed top-level `FlowVariables` block. A standard variable is one whose name is reserved. Standard variables are intended for flow-control and/or device binning. A tester running STIL.4 treats a standard variable as it would any other. A STIL.4 input stream shall not be required to define any of the standard variables. Figure 81 shows required standard variable attributes in bold. Variable attributes other than type and name are flexible, e.g., any of these variables could be defined with an alternate description, as a shared memory type, etc.

```
FlowVariables {
  Const String StdChipType = None {
    Permissions RhsReadWrite;
    ReInitAt LOAD;
    Description "Chip type identifier"; }
  Const String StdDeviceType = None {
    Permissions RhsReadWrite;
    ReInitAt LOAD;
    Description "Chip/pkg/wire-bond combination identifier"; }
  Const TestMode StdTestMode = None {
    Permissions RhsReadWrite;
    ReInitAt START;
    Description "Development, Engineering, or Production mode"; }
  Const String StdPackageType = None {
```

```
    Permissions RhsReadWrite;
    ReInitAt LOAD;
    Description "Package type identifier"; }
  Const Celsius StdTestTemperature = None {
    Permissions RhsReadWrite;
    ReInitAt LOAD;
    Description "Intended device test temperature"; }
}
```

Figure 81—Example: standard global variable definitions

Figure 81 shows tester independent definitions, hence the initializations to `None`. On a tester running STIL.4, a function call may be used to initialize a standard variable. If so, that function shall return the type of value required by that variable, e.g., a function used to initialize `StdChipType` shall return a value of type `String`.

`StdChipType`: represents the die. Its value may be `None` during package test.

`StdDeviceType`: represents the device or final product identifier, i.e., it may name the chip-type at wafer test or the chip-type/package-type/wire-bonding combination at package test.

`StdTestMode`: an enumerated type representing test mode. See 35.1 for the definition.

`StdPackageType`: represents the package type. Its value shall be `None` during wafer test.

`StdTestTemperature`: represents the temperature the device is intended to be tested at. If the temperature is unspecified, its value shall be `None`. Specifically, `StdTestTemperature` should not be used to represent the current chuck or junction temperature.

35.3 Flow control defaults (FlowExtended)

35.3.1 General

STIL.4 provides standard `FlowNode` and `TestBase` definitions using the syntax defined in Clause 30 and Clause 27, respectively. These definitions enable syntactical shortcuts for common stop-on-fail/adaptive-test behavior which make it easier to understand what a flow is doing by not becoming mired in repetitive details. Figure 76 shows the relationship between complete explicit `FlowNode` descriptions and syntactical shortcuts.

The user may provide defaults which shall appear in the input stream before any statement that invokes them. In practical terms, the safest place is before `TestType` or `FlowType` definitions.

35.3.2 Standard flow-node

The standard flow-node is capable of executing a test or flow and a set of actions that are designed, in concert with `TestBase` actions, to exhibit stop-on-fail behavior that is easily translated to target ATE. Stop-on-fail behavior is described as run-until-fail, then bin-and-stop, otherwise continue. Which of the actions that model this behavior are apportioned to `TestBase` and which are apportioned to the standard flow-node is governed by an organizational philosophy that attempts to make it easy to target the greatest number of known ATE. A provider may customize that apportionment to suit the behavior of a particular target tester.

The standard flow-node definition shall be accessed automatically by a STIL.4 compliant tool, i.e., it shall not require an `Include` statement. The standard flow-node shall be defined once before first use as an unnamed flow-node definition at the top-level, i.e., outside the scope of any flow, test-type or flow-type, the only other places where a flow-node may be defined. It is safest to define the standard flow-node before `TestType`, `Test`, `FlowType`, `Flow`, or `TestProgram`.

The following is the standard flow-node definition:

```
1 FlowNode {
2     PreActions {}
3     TestExec;
4     PostActions {}
5     ExitPorts {
6         Port True {} Next;
7     }
8 }
```

The user may override standard default flow-node `PreActions`, `PostActions`, and `ExitPorts` by explicitly specifying them in a default `FlowNode` block. Note that `TestExec` shall always be overridden by a `TestExec` that executes a user-specified test. The following is commentary on the standard flow-node definition:

— When a flow-node is instantiated, a copy of the default flow-node is made and its placeholder `TestExec` statement is replaced with one that refers to a test or flow.
— An implicit instantiation has no name and uses the default flow-node `PreActions`, `PostActions`, and `ExitPorts`. An explicit instantiation may have a name and may override the default's `PreActions`, `PostActions`, and/or `ExitPorts`.

35.3.3 Standard TestBase

The standard `TestBase` definition shall be accessed automatically by a STIL.4 compliant tool, i.e., it shall not require an `Include` statement. `TestBase` is an abstract base type, i.e., it cannot be instantiated. STIL.4 also specifies a standard `FlowNode`, enumerated types, and top-level variables necessary for the functioning of the standard test-types and flow-types.

The purpose of `TestBase` is to provide a common set of elements for all `TestType` and `FlowType` definitions. `TestBase` shall be defined only once in the input stream. `TestBase` requires the standard definition of enumerated type `FailMode` (see 35.1).

The standard configuration of `TestBase` executes a set of actions that are designed, in concert with `FlowNode` actions, to exhibit stop-on-fail behavior that is easily translated to target ATE. Stop-on-fail behavior is described as run-until-fail, then bin-and-stop, otherwise continue. Which of the actions that model this behavior are apportioned to `TestBase` and which are apportioned to the standard flow-node is governed by an organizational philosophy that attempts to make it easy to target the greatest number of known ATE. A provider may customize that apportionment to suit the behavior of a particular target tester.

In addition to syntactic form and semantics, a test-type or flow-type derived from `TestBase` also inherits concrete content. The minimum mandated content of `TestBase` is shown in Figure 82. A tool provider may define an alternate version with additional parameters and/or action blocks with differing content.[59]

[59] This is usually done in conjunction with defining a default `FlowNode` because the `TestBase` and default `FlowNode` definitions together are expected to produce desirable behavior when shorthand syntax is used to describe a flow.

```
TestBase {
  Parameters {
    Out   Const String   Type { ReInitAt LOAD; }  // Ref type()
    Out   Const String   Id   { ReInitAt LOAD; }  // Ref name()
    Out   Const String   App        = "" { ReInitAt LOAD; }
    Out         FailMode failMode   = NOEXEC { ReInitAt TEST_ENTRY; }
    In          Integer  testNumber = None { Optional; }
    InOut       BinSpec  failBin    = None { Optional; BinType Fail; }
    InOut       BinSpec  passBin    = None { Optional; BinType Pass; }
  }
  FlowVariables {}
  PreActions {}
  TestExec;
  PostActions {}
  PassActions {
    SetBin passBin;
  }
  FailActions {
    If (Local.failMode != NOEXEC) {
      Parent.failMode = Local.failMode;
      SetBinStop failBin;
    }
  }
}
```

Figure 82—Example: minimum content standard TestBase definition

A description of the required parameters follows:

Type: appears to be initialized to `None`; however, this parameter is unique in that it is automatically initialized to the test-type or flow-type name when the derived type is instantiated.[60]

Id: appears to be initialized to `None`; however, this parameter is unique in that it is automatically initialized to the derived type's instance name on instantiation.[61]

App: set to an empty string by default, this parameter may be used to indicate the specific application of this type of test or flow, e.g., when parameter `Type` is set to `"DCMeasurement"`, a fictional type for this example, the user may choose to initialize `App` to `"VOH"`.

failMode: set to `Enum FailMode` enumeration `PASS` when `TestExec` is executed and passes, set to appropriate alternate `FailMode` enumeration otherwise (see 35.1). `Enum FailMode` may be augmented to allow for user-defined fail codes. `PassActions` or `FailActions` are automatically selected as illustrated by the following pseudo-code:

`If (Local.failMode == PASS){` *execute PassActions* `} Else {` *execute FailActions* `}`

testNumber: a runtime read-only variable which is set at instantiation, normally used for tests but not flows.

failBin: any command using this variable is a no-op when set to `None`.

passBin: any command using this variable is a no-op when set to `None`.

[60] Dot notation access to parameter `Type` requires that the test be instantiated.
[61] For inline instantiations which are anonymous, Id is set to an empty string.

35.3.4 Flow control example

Figure 83 shows an example of FlowNode and Test interaction.

```
 1 TestType RequiredSetupTest {
 2   Inherit StdFunctional;
 3 }
 4
 5 Test RequiredSetupTest setup1 {
 6   testNumber = 1;
 7   patburst   = setupburst1;
 8   tim        = setupac;
 9   dclev      = setupdc;
10 }
11
12 Test RequiredSetupTest setup2 {
13   testNumber = 2;
14   patburst   = setupburst2;
15   tim        = setupac;
16   dclev      = setupdc;
17   failBin    = failSetup;
18 }
19
20 Flow StdFlow main {
21   TestExec {
22     FlowNode {
23       TestNumber 10;         // Overrides setup1 testNumber
24       PreActions {
25         If (skipThisTest)
26           Bypass;            // Resumes at PostActions
27       }
28       TestExec setup1;       // 1st attempt at setup
29       ExitPorts {
30         NO_EXEC: Port 'CurrentExec.failMode == NOEXEC' {}
31           Next SKIP1;        // Skip dependent tests
32         PASS:    Port 'CurrentExec.failMode == PASS' {}
33           Next DEPEND;       // Run dependent tests
34         FAIL:    Port True {}
35           Next;              // Try 2nd attempt at setup
36       }
37     }
38
39     FlowNode {               // Uses setup2 testNumber
40       PreActions {
41         If (skipThisTest)
42           Bypass;            // Resumes at PostActions
43       }
44       TestExec setup2;       // 2nd attempt at setup
45       ExitPorts {
46         PASS: Port 'CurrentExec.failMode == PASS' {}
47           Next DEPEND;       // Run dependent tests
48         FAIL: Port True {}
49           Next SKIP1;        // Skip dependent tests
50       }
51     }
52   }
53
```

```
54   FlowNode DEPEND { TestExec flow1; }
55
56   FlowNode SKIP1 { TestExec flow2; }
57 }
```

Figure 83—Example: FlowNode/Test interaction

The premise of flow `main` in Figure 83 is that it either executes flow1 following successful configuration of the DUT or skips to flow2 which does not depend on device configuration. `flow1` and its associated configuration tests may be skipped under control of the Boolean variable `skipThisTest` or via some adaptive test process (not described here). If executed, flow `main` makes up to two attempts to configure the DUT as controlled by Boolean variable `skipThisTest`. If the second attempt is made and fails, soft bin `failConfig` is set and the program stops. If no attempts are made, the flow carries on with `flow2` at flow-node `SKIP1`.

Figure 83 employs standard `FlowNode` and `TestBase` definitions. Identifiers `failSetup`, `setupburst1`, `setupac`, `setupdc`, `setupburst2`, `skipThisTest`, `flow1`, and `flow2` are presumed to have been defined before use.

The following line comments refer to Figure 83:

- Lines 1–3: defines `RequiredSetupTest` as a type of `StdFunctional` test to indicate intent.
- Lines 5–10: defines an instance of `RequiredSetupTest` named `setup1`. Unmentioned parameters take on default values defined in standard `TestBase`, `StdFunctional`, and theoretically `RequiredSetupTest` defined in this example, which has no parameters of its own.
- Lines 12–18: defines an instance of `RequiredSetupTest` named `setup2`. Other than the name, the commentary for lines 5–10 applies.
- Lines 20–57: defines an instance of `StdFlow` named `main`. As required, it replaces the standard flow-node `TestExec` with its own.
- Lines 22–37: defines an anonymous flow-node. Like all flow-nodes under FlowExtended, instantiation begins by copying the standard flow-node. This particular instantiation then overrides the standard flow-node `TestNumber`, `PreActions`, `TestExec` (which every flow-node instantiation is required to override), and `ExitPorts`.
- Line 26: `PostActions` occur between `TestExec` and `ExitPorts`. Because they are not specified in this flow-node, they are as specified in the standard flow-node, i.e., an empty set.
- Lines 30–31: This covers the situation where `setup1` is not executed via a mechanism not described by STIL.4 which intercepts the execution of `setup1`, e.g., intervening adaptive test software, then dependent tests are skipped.
- Lines 39–51: defines an anonymous flow-node in a manner similar to that described for lines 22–37.
- Line 54: defines a flow-node named `DEPEND`. This flow-node uses the `PreActions`, `PostActions`, and `ExitPorts` defined in the standard flow-node. Note that attribute `TestNumber` is not set in the standard flow-node and therefore not set in this instantiation. The syntactical shortcut

  ```
  TestExec flow1;
  ```

  could replace this line.
- Line 56: defines a flow-node named `SKIP1`. Other than the name and `TestExec`, the commentary for line 54 applies.

35.4 Standard No-op and None (FlowExtended)

Definitions of standard test-types shall be accessed automatically by the STIL.4 compliant tool, i.e., they shall not require an `Include` statement.

```
TestType None{
    Inherit TestBase;
    PreActions {}
    TestExec;
    PostActions {}
    PassActions {}
    FailActions {}
}
```

Test-type `None` shall inherit all its parameters from `TestBase`, have no local variables, and override all `TestBase` actions with actions that do nothing. Its `TestExec` function shall do nothing. This facilitates entry-point initialization to `None`, e.g.:

```
  On LOAD None;
```

The above code instantiates test-type `None` inline. Test-type `None` shall only be used as keyword `None` and as a base type for standard test-type `StdNoOp` as follows:

```
TestType StdNoOp {
    Inherit None;
}
```

Test-type `StdNoOp` may serve as a place-holder (inline instantiation) or as a base type from which to derive other test-types that may define parameters and/or local variables and execute pre- and/or post-actions.

35.5 Standard PatternExec test (FlowExtended)

The definition of `StdPatternExec` shall be accessed automatically by a STIL.4 compliant tool, i.e., it shall not require an `Include` statement.

`StdPatternExec` is a standard predefined `TestType` from which a `Test` can be instantiated. A `StdPatternExec` test loads level and timing information and subsequently executes one or more patterns. It shall not be used to load only levels or timing or only run patterns.

```
TestType StdPatternExec {
    Inherit TestBase;
    Parameters {
        InOut PatternExec patexec;
    }
    TestExec;
}
```

Parameters:

`patexec`: this is a reference to a previously defined `PatternExec` block. The contents of that block shall adhere to the requirements described in the next paragraph. Passing a reference to a

PatternExec that refers to an empty Timing, DCLevels, or PatternBurst shall result in undefined behavior.

In addition to the syntax and semantics described in STIL.0, STIL.2, and clarifications described under Clause 10 in this document, a PatternExec block used as a StdPatternExec parameter shall meet the following requirements:[62]

— PatternBurst shall be specified for every instance of StdPatternExec.
— DC levels shall be specified for all signals exercised via associated patterns for the first StdPatternExec to be executed. DC levels shall be optional thereafter.
— Timing shall be specified for all signals exercised via associated patterns for every instance of StdPatternExec.

35.6 Standard functional test (FlowExtended)

The definition of StdFunctional shall be accessed automatically by a STIL.4 compliant tool, i.e., it shall not require an Include statement. See Figure 84.

StdFunctional is a predefined standard TestType from which a functional Test can be instantiated. Similar to StdPatternExec, a StdFunctional test loads level and timing information and subsequently executes one or more patterns. It has additional capabilities described by way of its parameters.

```
TestType StdFunctional {
    Inherit TestBase;
    Parameters {
        InOut       PatternBurst patburst;
        InOut Const Timing       tim;
        InOut Const DCLevels     dclev;
        InOut Const DCSets       dcsets   { Optional; }
        InOut       Category     accat    { Optional; }
        InOut       Category     dccat    { Optional; }
        InOut Const Selector     acsel    { Optional; }
        InOut Const Selector     dcsel    { Optional; }
        In          VecLocation  start    { Optional; }
        In          VecLocation  stop     { Optional; }
        In          Window       win      { Optional; }
        In          Seconds      maxtime  { Optional; }
        InOut       DCSequence   seqbeg   { Optional; }
        InOut       DCSequence   seqend   { Optional; }
    }
    TestExec;
}
```

Figure 84—Example: standard functional test definition

TestExec: runs non-STIL.4 functional test code using at minimum, required parameters patburst, tim, and dclev. Pre and post actions are inherited from TestBase. Sets TestBase parameter failMode to the appropriate value of type FailMode, i.e., to FAIL_FNC when the test fails, to PASS otherwise. Parameter details follow:

[62] This is an amalgamation of STIL.0 and STIL.2 rules amended by STIL.4 for the purpose of using PatternExec as a StdPatternExec parameter.

`accat`: contains spec variables for use with timing. Needed to resolve variable names used in test parameter `tim`.

`acsel`: required if test parameter `accat` uses values other than `Typ`, i.e., `Meas`, `Min`, or `Max`.

`dccat`: contains spec variables for use with DC levels. Needed to resolve variable names used in test parametes `dclev` and `dcsets`.

`dclev`: contains DC level information for `In`, `Out`, `InOut`, and `Supply` signals, that in conjunction with test parameters `tim` and `patburst` complete the information required to generate waveforms. See STIL.2 for more information.

`dcsel`: required if test parameter `dccat` uses values other than `Typ`, i.e., `Meas`, `Min`, or `Max`.

`dcsets`: contains a set of `DCLevels` that may be referenced in each pattern in test parameter `patburst` to change levels on the fly. Before the first such reference, the levels specified by test parameter `dclev` are in force.

`maxtime`: limits the maximum execution time. Terminates infinite or match loop patterns.

`patburst`: if test parameters `start` and `stop` are set to `None`, usually the default, then all `PatternBurst` and or `PatList Start` and `Stop` specifications are honored.

`seqbeg`: optional user-defined `DCSequence` to be executed before any levels are set by `TestExec`.

`seqend`: optional user-defined `DCSequence` to be executed after `TestExec`.

`start`: overrides any and all `Start` statements found within test parameter patburst. The location shall be unique in the set of patterns referenced in the `PatList`. Setting `start` to a location on or inside a loop or `MatchLoop` statement results in undefined behavior.

`stop`: overrides any and all `Stop` statements found within test parameter `patburst`. The location shall be unique in the set of patterns referenced in the `PatList`. Setting `stop` to a location on or inside a loop or `MatchLoop` statement results in undefined behavior.

`tim`: contains waveform character associated timing and waveform descriptions to be used in conjunction with test parameter `patburst` to generate logical waveforms.

`win`: describes a pattern window, i.e., vector range and signals, over which strobes specified in the timing and patterns shall be active. When the value of `win` is `None`, strobes specified in the timing shall be active over the entire `PatternBurst`, otherwise the vector range should fall between vector locations specified by parameters `start` and `stop` or `PatternBurst`/`Pattern Start` and `Stop` statements in order to be effective.

35.7 Standard flow (FlowExtended)

The definition of `StdFlow` shall be accessed automatically by a STIL.4 compliant tool, i.e., it shall not require an `Include` statement.

Flow-type `StdFlow` is used to instantiate a flow of tests with standard flow pre- and post-actions. The standard definition is as follows:

```
FlowType StdFlow {
    Inherit TestBase;
}
```

Annex A

(informative)

Event sequence

A.1 General

This annex describes the sequence of events as they occur on a virtual machine or actual ATE running STIL.4. The purpose of this clause is to unambiguously communicate the intent behind STIL.4, STIL.0, and STIL.2, especially as it relates to variable memory allocation and content. It is understood that when STIL.4, STIL.0, and STIL.2 code is translated to run on a particular tester, it may not be possible to wholly comply with the standard's intent. Understanding STIL.4 intent helps make test programs translated to run on different testers behave as consistently as possible.

A.2 Parsing and loading

Each test-program shall have its own copy of unamed and named `FlowVariables` referenced by the `TestProgram` block. Note that the reference to the unnamed `FlowVariables` block is implicit and precedes references to named `FlowVariables` blocks.

Variables defined inside a test or flow `FlowVariables` and `Parameters` block are initialized upon instantiation of that test or flow.

All tests or flows, named or anonymous/inline, are instantiated before any are executed. This may affect a top-level variable value, i.e., a test or flow may associate a default value with an `InOut` parameter that references a top-level variable. The value of that top-level variable may subsequently be employed during the instantiation of another test or flow.

Once all program information is loaded, a `LOAD` event is generated.

A.3 Execution

An asynchronous event shall be required to begin execution (see `AsynchronousEvent` in 35.1). STIL.4 defined asynchronous events shall occur in the following order: `LOAD`, `LOT_START`, `WAFER_START`, `START`, and then `RETEST`. When an asynchronous event is triggered:

1. Pre `TestExec` actions:
    a. Variables whose `ReInitAt` attribute matches the event name are initialized. This includes spec variable meas values, parameters and variables in `FlowVariables` blocks associated with `TestProgram` and `FlowVariables` blocks local to tests.
    b. If the event is `START`, then all soft and hard bins are unset and event-specific bin-associated counters are reinitialized.
2. The test or flow associated with the event, as specified under `EntryPoints`, is executed.
3. Post `TestExec` actions:
    a. If the event is `START`, then all soft bins are mapped to hard bins according to `BinMap` instructions.

Annex B

(informative)

Top-level block sequence (FlowExtended)

B.1 General

The intent of this clause is to provide an example of a safe block sequence, first in the form of a top-level skeleton, then in the form of a complete test program.

B.2 Skeleton and dependencies

This subclause provides a usable top level block sequence and the dependencies upon which this sequence is based. Standard definitions and extensions or modifications thereof should be read before any of the user supplied blocks.

Not all the blocks listed here need to be present in one input stream. Pattern related blocks for example may be separate. Some of the dependencies while possible or even likely may not be present under specific circumstances.

STIL.0/1/2/4 block	Dependencies
`STIL 1.0`	None
`Header`	None
`UserFunctions`	None
`Variables` [a]	None
`FlowVariables`	Variables (determines which variables are in shared memory)
`FlowVariables` VARS_NAME	Unnamed FlowVariables block
`Signals`	None
`SignalGroups`	Unnamed Signals block
`Signals` SIGS_NAME	Unnamed Signals block
`SignalGroups` GRPS_NAME	Signals block of the same name and unnamed Signals block
`Package` PKG_NAME	None (may be supplied via ATPRG library)
`Chip` CHIP_NAME	Signals, SignalGroups
`DCSequence` DCSEQ_NAME	Chip, Signals, SignalGroups
`Device` DEV_NAME	Chip, Package, Signals, SignalGroups, TestProgram,[b] DCSequence
`Spec` SPEC_NAME	None
`Selector` SEL_NAME	None
`Timing`	Device, Chip, Signals, SignalGroups
`Timing` TIM_NAME	Device, Chip, Signals, SignalGroups
`DCLevels`	Device, Chip, Signals, SignalGroups
`DCLevels` LEVELS_NAME	Device, Chip, Signals, SignalGroups
`DCSets` DCSET_NAME	DCLevels
`PatternBurst` BURST_NAME	Device, Chip, Signals, SignalGroups, MacroDefs, Procedures, ScanStructures, PAT_NAME or BURST_NAME

STIL.0/1/2/4 block	Dependencies
`PatternExec` PATEXEC_NAME	Spec, Category, Selector, DCLevels, DCSets, Timing, PatternBurst used by an automatic test pattern generator to generate StdFunctional test
`SoftBinDefs`	None
`SoftBinDefs` SOFTDEFS_NAME	None
`HardBinDefs`	None
`HardBinDefs` HARDDEFS_NAME	None
`BinMap` BINMAP_NAME	SoftBinDefs, HardBinDefs
`TestType` TESTTYPE_NAME [c]	TestBase and other base types, FlowVariables, PatternBurst, PatternExec
`FlowType` FLOWTYPE_NAME [d]	Same as TestType
`Test` TESTTYPE_NAME TEST_NAME	FlowVariables, BinMap, Timing, DCLevels, DCSets, Spec, Selector, DCSequence, PatternBurst, PatternExec
`Flow` FLOWTYPE_NAME FLOW_NAME	Same as Test
`SignalMap` SIG_MAP_NAME	Unnamed Signals block
`TestProgram` PGM_NAME	Variables, FlowVariables, BinMap, SignalMap,[e] Test and Flow types
`ScanStructures` SCAN_NAME	Signals
`Procedures`	None
`Procedures` PROCS_NAME	None
`MacroDefs`	None
`MacroDefs` MDEFS_NAME	None
`Pattern` PAT_NAME	Device, Chip, Signals, SignalGroups, Procedures, MacroDefs, ScanStructures

[a] This block is expected to be pre-populated with standard global variables (see 35.2).

[b] Forward reference.

[c] Standard types are expected to be defined before user code.

[d] Standard types are expected to be defined before user code.

[e] There is no `SignalMap` dependency if the `Device` block is used instead of the `SignalMap`.

`PatternExec` is not directly used by STIL.4. An ATPRG may use it to generate a `TestType` instance using keyword `Test`. Pattern code is usually kept in separate files.

Annex C

(informative)

Usage examples (FlowExtended)

C.1 Coding examples

This section contains sample coding that helps illustrate language features used in concert for specific applications.

C.1.1 Speed binning with scalable timing

The code in this subclause combines the following elements:

- Timing that consists in part or whole of timing expressions that are a function of period
- Passing bins on a speed bin axis whose names reflect device speed, e.g., "900MHz", "600MHz", and "300MHz"
- Functions `str2number` and `eval`
- `TestType`

This technique allows the user to easily add, subtract, or temporarily enable or disable speed tests by respectively adding, subtracting, or enabling/disabling soft bins on the speed axis. When adding or subtracting, soft to hard bin mapping must be synchronized.

```
// ===================================================================
Signals {       // Set-reset latch
  Set   In;
  Reset In;
  Q     Out;
  QBar  Out;
  Vdd   Supply;
  Gnd   Ground; // STIL.4 property
}
// -------------------------------------------------------------------
SignalGroups {            // Set-reset latch
  DIns = 'Set + Reset';   // Digital inputs
  DOuts = 'Q + QBar';     // Digital outputs
  DSigs = 'DIns + DOuts'; // Digital signals
}
// ===================================================================
Pattern pat1 {
  WaveformTable one;
  V { DSigs=10HL; }
  V { DSigs=00HL; }
  V { DSigs=01LH; }
  V { DSigs=00LH; }
  V { DSigs=11XX; }
}
// -------------------------------------------------------------------
PatternBurst Burst1 {
```

```
  PatList { pat1; }
}
// ============================================================================
Spec SpecSpeed {
  Category CatPeriod {
    period { Units "s"; } // STIL.4: init to None, constrain to Seconds
  }
}
// ----------------------------------------------------------------------------
Selector SelPeriod {
  period Meas;
}
// ============================================================================
Timing AcLoose {
  WaveformTable one {
    Period 'period';
    Waveforms {
      DIns { 01 { '0s' D/U; }}
      DOuts {
        01 { '0s' D/U; }
        LH { 'period*0.9' L/H; }
        X { '0s' X;}
      }
    }
  }
}
// ============================================================================
DCLevels LowV {
  Vdd {
    VForce '3.3V';
    IClamp '8mA';
  }
  DIns {
    VIH  '2.0V';
    VIL  '0.8V';
  }
  DOuts {
    VIH  '2.0V';
    VIL  '0.8V';
    VOH  '2.4V';
    VOL  '0.4V';
    IOH  '3mA';
    IOL  '3mA';
    LoadVRef '1.8V';
  }
}
// ============================================================================
SoftBinDefs softbindefs {
  Pass {
    BinAxis Speed {
      Bin "900MHz";      // Index 0, Number 1
      Bin "600MHz";      // Index 1, Number 2
      Bin "300MHz";      // Index 2, Number 3
    }
  }
  Fail {
    Bin Functional;
```

```
  }
}
// ------------------------------------------------------------------
HardBinDefs hardbindefs {
  // Explicit bin numbers are the same as default
  Pass {
    Bin "900MHz"       { Number 1; }                        // Index 0
    Bin "600MHz"       { Number 2; }                        // Index 1
    Bin "300MHz"       { Number 3; }                        // Index 2
    Bin Unmarketable   { Number 4; Verbose "Too slow"; }    // Index 3
    Bin Unclassifyable { Number 5; Verbose "No bins set"; } // Index 4
  }
  Fail {
    Bin Funct          { Number 6; }                        // Index 0
  }
}
// ------------------------------------------------------------------
BinMap binmap {
  SoftBinDefs softbindefs;
  HardBinDefs hardbindefs;

  // Maps soft bin to hard bin number
  Map Pass.Bins["900MHz"]   -> 1;
  Map Pass.Bins["600MHz"]   -> 2;
  Map Pass.Bins["300MHz"]   -> 3;
  Map None                  -> 5;  // Unclassifyable
  Map Fail.Bins[Functional] -> 6;  // Fail Timing
}
// ==================================================================
// APPLICATION: functional testing at multiple speeds when the timing
// over the speed range scales to period.

// DESCRIPTION: Iterate over bins on softbin axis "passBin", beginning
// at index 0 executing a StdFunctional test with each iteration. Set
// pass bin corresponding to category on first pass and exits. Set fail
// bin if none of the iterations pass.

// REQUIREMENTS:
//   - Parameter "passBin" is a BinAxis whose bin names are test
//     speeds in units of Hz, ordered fast to slow.
//
//   - Speeds to be tested have their corresponding bins enabled.
//
//   - "prd" is a reference to the spec-variable in "accat" that
//     represents the period.
//
//   - "acsel" selects the Meas field for the parameter passed in as
//     "prd".
//
//   - Some or all of the timing in "tim" is expressed in terms of a
//     spec-variable in "accat" that represents the period.
TestType ScalableSpeed {
  Inherit TestBase {
    passBin = None { BinType Axis; }             // Add constraint
  }
  Parameters {
    InOut       PatternBurst patburst;
```

```
  InOut Const Timing       tim;
  InOut Const DCLevels     dclev;
  InOut       Category     accat    { Type Timing; }
  InOut       Category     dccat    { Optional; Type DCLevels; }
  InOut Const Selector     acsel;
  InOut Const Selector     dcsel    { Optional; }
  InOut       SpecVariable prd      { Units "s"; }
}
FlowVariables {
  Const Integer bcount = passBin.Bins.size();   // Axis bin count
  Integer idx    = 0;                           // Index for axis
  Hertz   speed  = None;
  String  bname  = passBin[idx].name();
  Seconds period = 1/speed;
}
PreActions {
  While (!passBin[idx].isEnabled()) {
    idx = idx + 1;
    If (idx == bcount) {
    Bypass; // Skip TestExec and post-actions
    }
  }

  If (str2number(bname, speed) == bname.size())
    prd.Meas = eval(period);  // Evaluate then assign tracking period
  Else {
    Stop;
  }
}
// ---------------------------------------------------------------------
// The following TestExec statement theoretically creates an
// anonymous instance of StdFlow and StdFunctional when TestType
// ScalableSpeed is instantiated. The StdFlow instance contains 1
// default FlowNode which executes StdFunctional.
TestExec StdFunctional {
  // The following assignments occur only once at TestType
  // StdFunctional/ScalableSpeed instantiation
  Local.patburst = &Parent.patburst;
  Local.tim      = &Parent.tim;
  Local.dclev    = &Parent.dclev;
  Local.accat    = &Parent.accat;
  Local.dccat    = &Parent.dccat;
  Local.acsel    = &Parent.acsel;
  Local.dcsel    = &Parent.dcsel;
}
// ---------------------------------------------------------------------
PassActions {
  SetBin passBin[idx];              // Set soft bin
}
FailActions {
  If (idx == bcount) {              // Exhausted number of speed bins
    Local.failMode = FAIL_PRM;      // TestBase pass/fail indicator
    SetBin Local.failBin;           // Soft bin
  } Else {
    idx = idx + 1;                  // Increment to bin-axis next bin
    If (str2number(bname, speed) == bname.size()) { // Get next speed
      prd.Meas = eval(period);      // Variable period is tracking
```

```
        ReExec;                        // Loop back to TestExec
      } Else {
        Stop;
      }
    }
  }
}
// ===================================================================
TestProgram pgm {
  BinMap binmap;               // Refer to BinMap
  Test ScalableSpeed speed {  // Instantiate test
    passBin  = &Pass.Speed;    // Pass group Speed axis
    failBin  = &Functional;
    patburst = &Burst1;
    tim      = &AcLoose;
    accat    = &SpecSpeed.CatPeriod;
    acsel    = &SelPeriod;
    dclev    = &LowV;
  }
  Flow StdFlow mainflow {
    TestExec speed; // Default FlowNode reference to test instantiation
  }
  EntryPoints {
    On Start mainflow;
  }
}
// ===================================================================
```

C.1.2 FlowExtended production test program example

The program example in Figure C.2 performs a series of functional tests, setting a soft pass or fail bin with exit-on-fail behavior. After exiting, a hard bin is set according to bin-map specifications. The program is based on the fictional device shown in Figure C.1 where input `C` controls the output levels on `Z`. Patterns are usually in a separate input stream, i.e., set of files, but for this small example are put in line with the rest of the information.

This example employs syntax enabled by the `FlowExtended` mode including the `TestType StdFunctional` and standard `TestBase` components.

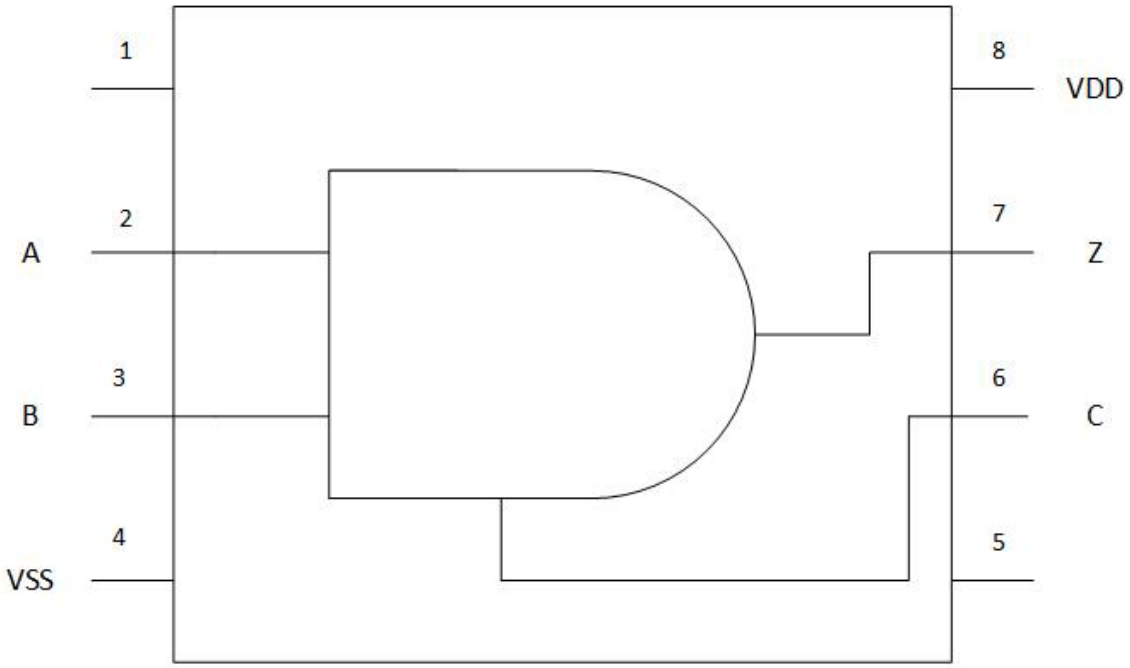

Figure C.1—Diagram: And gate with programmable output levels

```
// ================================================================
STIL 1.0 {
  FlowExtended 2017;
  DCLevels 2002;
}
// ================================================================
Header {
  Title "Basic Production Test Program";
  Date "Tue Dec  3 17:34:18 EST 2013";
  Source "ExBasicProdPgm.stil";
  History {
      Ann {* 12/4/2013 - made a change *}
  }
}
// ================================================================
// User-defined variables
FlowVariables {
    Boolean tightFncFailed = False
        { Description "True if any test of type TightFnc failed"; }
}
// ================================================================
// Single 2 input AND gate with programmable output levels
// Signal Subtype is Digital by default
Signals {
    VDD Supply;
    VSS Ground;
    A In;
    B In;
    C In;                // Control: 1 = 5V levels, 0 = 3.3V levels
    Z Out;
}
```

```
32 // ----------------------------------------------------------------
33 SignalGroups {
34     INPUTS  = 'A + B + C';
35     OUTPUTS = 'Z';
36     ALL     = 'INPUTS + OUTPUTS';
37 }
38 // ----------------------------------------------------------------
39 Signals B4081_IN_DIP_8P {
40     OPIN1 InOut + Open;
41     OPIN5 InOut + Open;
42 }
43 // ----------------------------------------------------------------
44 SignalGroups B4081_IN_DIP_8P {
45     OPENS = 'OPIN1 + OPIN5';
46 }
47 // ================================================================
48 // 16 pin dual inline package, top view:
49 //   - 1 x    x 8 VDD
50 //   A 2 x    x 7 Z
51 //   B 3 x    x 6 C
52 // VSS 4 x    x 5 -
53 Package DIP_8P {
54     PinList { 1..8 }
55     Plane PWR { 8 }
56     Plane GND { 4 }
57 }
58 // ================================================================
59 Chip B4081;// Chip defined by unnamed Signals & SignalGroups blocks
60 // ================================================================
61 // Chip in an 8 pin DIP
62 Device B4081_IN_DIP_8P {
63     Chip B4081;
64     Package DIP_8P;
65     Signals      B4081_IN_DIP_8P;
66     SignalGroups B4081_IN_DIP_8P;
67     PinMap {
68         // Signal-to-package-pin or -plane mapping
69         VDD      PWR;
70         VSS      GND;
71         A        2;
72         B        3;
73         C        6;
74         Z        7;
75         OPIN1    1;
76         OPIN5    5;
77     }
78     Tester " Teradyne, Ultraflex " {
79         // Fictional tester defaults to a generic Configuration
80         TestHead {
81             Partition {
82                 TestProgram production;
83                 DeviceSites 1 {
84                     ChannelMap {
85                     //  Signal-to-channel mapping
86                         VDD       23.sense1;
87                         VSS       Gnd;
88                         A         4.ch1;
```

```
 89                          B         4.ch12;
 90                          C         4.ch13;
 91                          Z         4.ch14;
 92                          OPIN1     4.ch15;
 93                          OPIN5     4.ch16;
 94                          }
 95                      }
 96                  }
 97              }
 98      }
 99 }
100 // ================================================================
101 Timing acloose {
102     WaveformTable wft {
103         Period '50ns';
104         Waveforms {
105             INPUTS {
106                     01 { '0ns' ForceDown/ForceUp; }
107                 }
108             OUTPUTS {
109                 HLZ {
110                     '0ns'  ForceOff;
111                     '25ns' CompareHigh/CompareLow/CompareOff;
112                 }
113             }
114         }
115     }
116 }
117 // ----------------------------------------------------------------
118 Timing actight {
119     WaveformTable wft {
120         Period '25ns';
121         Waveforms {
122             INPUTS {
123                     01 { '0ns' ForceDown/ForceUp; }
124                 }
125             OUTPUTS {
126                 HLZ {
127                     '0ns'    ForceOff;
128                     '12.5ns' CompareHigh/CompareLow/CompareOff;
129                 }
130             }
131         }
132     }
133 }
134 // ================================================================
135 DCLevels dctight5V {
136     VDD {
137         VForce '5.0V';
138         IClamp '50mA';
139     }
140     INPUTS {
141         VIH '3.7V';
142         VIL '1.3V';
143     }
144     OUTPUTS {
145         VOH '4.7V';
```

```
146         VOL '0.2V';
147         IOH '-10mA';
148         IOL '10mA';
149         LoadVRef '2.5V';
150     }
151 }
152 // ----------------------------------------------------------------
153 DCLevels dctight3V {
154     VDD {
155         VForce '5.0V';
156         IClamp '50mA';
157     }
158     INPUTS {
159         VIH '3.7V';
160         VIL '0.8V';
161     }
162     OUTPUTS {
163         VOH '2.4V';
164         VOL '0.5V';
165         IOH '-10mA';
166         IOL '10mA';
167         LoadVRef '1.5V';
168     }
169 }
170 // ================================================================
171 PatternBurst burst5V {
172     PatList { pat5V; }
173 }
174 // ----------------------------------------------------------------
175 PatternBurst burst3V {
176     PatList { pat3V; }
177 }
178 // ================================================================
179 SoftBinDefs softbindefs {
180     Pass {
181         Bin Passed;     // Index 0, Number 1
182     }
183     Fail {
184         Bin LooseFunct; // Index 0, Number 2
185         Bin TightFunct; // Index 1, Number 3
186     }
187 }
188 // ----------------------------------------------------------------
189 HardBinDefs hardbindefs {
190   Pass {
191     Bin Passed {                            // Index 0, Number 1
192         WafermapChar *;
193     }
194     Bin NoBin {                             // Index 1, Number 2
195         WafermapChar ?;
196         Verbose "No soft bin set";
197     }
198   }
199   Fail {
200     Bin TightFunct {                        // Index 0, Number 3
201         WafermapChar T;
202         Verbose "Tight timing, tight levels";
```

```
203     }
204     Bin LooseFunct {                          // Index 1, Number 4
205         WafermapChar L;
206         Verbose "Loose timing, tight levels";
207     }
208   }
209 }
210 // ----------------------------------------------------------------
211 BinMap binmap {     // Soft to hard bin mapping
212   SoftBinDefs softbindefs;
213   HardBinDefs hardbindefs;
214
215   // Sends device to this physical bin when no sort bins are set:
216   Map None                  -> 2; // Unclassifyable
217
218   Map Pass.Bins[Passed]     -> 1;
219
220   Map Fail.Bins[TightFunct] -> 3;
221   Map Fail.Bins[LooseFunct] -> 4;
222 }
223 // ================================================================
224 TestType LooseFnc {
225     Inherit StdFunctional {
226         failBin = LooseFunct; // Override default fail bin
227         passBin = Passed;     // Override default pass bin
228     }
229     PreActions {               // Override standard PreActions
230         If (tightFncFailed == False)
231             Bypass;
232     }
233 }
234 // ----------------------------------------------------------------
235 TestType TightFnc {
236     Inherit StdFunctional {
237         failBin = TightFunct; // Override default fail bin
238         passBin = Passed;     // Override default pass bin
239     }
240     FailActions {             // Override standard FailActions
241         tightFncFailed = True;                // Override
242         Parent.failMode = Local.failMode;     // Standard
243         SetBinStop failBin;                   // Standard
244     }
245 }
246 // ================================================================
247 TestProgram production {
248     // Uses top-level unnamed FlowVariables block only
249     BinMap binmap;
250     Test LooseFnc tstLf5V {
251         patburst = burst5V;   // Forward reference OK
252         tim      = acloose;
253         dclev    = dctight5V;
254     }
255     // ------------------------------------------------------------
256     Test LooseFnc tstLf3V {
257         patburst = burst3V;   // Forward reference OK
258         tim      = acloose;
259         dclev    = dctight3V;
```

```
260     }
261     Test TightFnc tstTf5V {
262         patburst = burst5V;   // Forward reference OK
263         tim      = actight;
264         dclev    = dctight5V;
265     }
266     Test TightFnc tstTf3V {
267         patburst = burst3V;   // Forward reference OK
268         tim      = actight;
269         dclev    = dctight3V;
270     }
271     // -------------------------------------------------------------
272     Flow StdFlow floLoose {
273         TestExec tstLf5V;           // Implicit standard flow-node
274         TestExec tstLf3V;           // Implicit standard flow-node
275     }
276     Flow StdFlow floTight {
277         TestExec tstTf5V;           // Implicit standard flow-node
278         TestExec tstTf3V;           // Implicit standard flow-node
279     }
280     Flow StdFlow floMain {
281         TestExec floLoose;          // Implicit standard flow-node
282         TestExec floTight;          // Implicit standard flow-node
283     }
284     // -------------------------------------------------------------
285     EntryPoints {
286         On START floMain;
287     }
288 }
289 // =============================================================
290 Pattern pat5V {
291     WaveformTable wft;
292     // Signal ABCZ
293     V { ALL = 001L; }
294     V { ALL = 011L; }
295     V { ALL = 101L; }
296     V { ALL = 111H; }
297 }
298 // -------------------------------------------------------------
299 Pattern pat3V {
300     WaveformTable wft;
301     // Signal ABCZ
302     V { ALL = 000L; }
303     V { ALL = 010L; }
304     V { ALL = 100L; }
305     V { ALL = 110H; }
306 }
307 // =============================================================
```

Figure C.2—Example: small production test program

Annex D

(informative)

Switching from Flow to FlowExtended

STIL.4 provides for two tiered capability. One is invoked by keyword `Flow`, the other by keyword `FlowExtended`, in the `STIL` statement block (Clause 7). `FlowExtended` syntax is mostly a superset of `Flow`. There are notable exceptions:

- In Flow mode, a Test instance has no relationship to TestBase whereas in FlowExtended mode, a Test instance does.
- In Flow mode, a Flow instance has no relationship to TestBase whereas in FlowExtended mode, a Flow instance does.

Recommendations for `Flow` mode in anticipation of switching to `FlowExtended`:

- Avoid using names reserved for `FlowExtended` under 35.1.3 when defining enumerated types.
- Avoid using names reserved for `FlowExtended` under 35.2 when defining global variables unless used for the same purpose.[63]
- Avoid using `TestBase` parameter names (35.3.3) as `TestMethod` parameter names.[64]

To use `FlowExtended` capabilities, a user or software vendor may need to make some changes:

- `TestMethod` based tests work in `FlowExtended` mode. To take full advantage of the `FlowExtended` capability of changing or augmenting the behavior of an existing `TestType` or `FlowType` via inheritance or combining existing test-types to create a new one, a `TestMethod` should be replaced by a comparable `TestType`.[65] Note that a `Test` instance is created by referencing a `TestMethod` in `Flow` mode, the test-method being defined outside the scope of STIL.4, or by instantiating a `TestType` that is wholly or partially defined using STIL.4 `FlowExtended` syntax.
- Depending on the software vendor, `Flow` and `FlowExtended` mode may behave differently with regard to test or flow pass/fail status specification and propagation. Default behavior in `FlowExtended` mode uses `TestBase` (35.3.3) parameter `failMode` (type `FailMode` in 35.1.3) to bubble status up the containment hierarchy until it reaches the test-program entry-point level.

[63] A translator may attach a specific meaning to the identifier.
[64] `FlowExtended` depends on a flow or test exhibiting `TestBase` properties; therefore, a test instantiated via `TestMethod` should adopt `TestBase` properties and a `Flow` instantiated without specifying the `FlowType` should imply type `StdFlow`.
[65] Replacing a `TestMethod` with a `TestType` requires using type `SignalGroup` instead of `sigref_expr` (these types serve essentially the same purpose).

Consensus

WE BUILD IT.

Connect with us on:

Facebook: https://www.facebook.com/ieeesa

Twitter: @ieeesa

LinkedIn: http://www.linkedin.com/groups/IEEESA-Official-IEEE-Standards-Association-1791118

IEEE-SA Standards Insight blog: http://standardsinsight.com

YouTube: IEEE-SA Channel

IEEE
standards.ieee.org
Phone: +1 732 981 0060 Fax: +1 732 562 1571