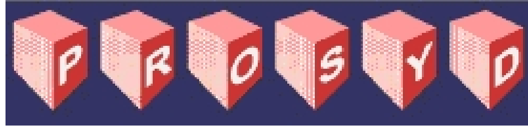

FP6-IST-507219

PROSYD:

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

"Property-by-Example" Guide: a Handbook of PSL Examples (Deliverable 1.1/3)

Due date of deliverable: May 1, 2005 (Reissue)
Actual submission date: May 1, 2005

Start date of project: January 1, 2004 Duration: Three years

Organisation name of lead contractor for this deliverable: IBM

Revision 2.0

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	☒
PP	Restricted to other programme participants (including the Commission Services)	☐
RE	Restricted to a group specified by the consortium (including the Commission Services)	☐
CO	Confidential, only for members of the consortium (including the Commission Services)	☐

Notices

For information, contact Avigail Orni, ornia@il.ibm.com.

This document is intended to fulfil the contractual obligations of the PROSYD project concerning deliverable 1.1/3 described in contract number 507219.

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

© Copyright PROSYD 2005. All rights reserved.

Table of Revisions

Version	Date	Description and reason	By	Affected sections
1.0	August 31, 2004	Final revision for delivery	A. Orni	All
1.1	March 20, 2005	Updated to comply with new template	A. Orni	All
1.2	April 5, 2005	Modified as per review comments of October 2004 (draft for review by partners)	A. Orni	All
1.3	April 25, 2005	Incorporated review comments received from partners	A. Orni	All
2.0	April 28, 2005	Final approval by project coordinator	A. Orni for C. Eisner	All

Authors

Shoham Ben-David
Avigail Orni

Executive Summary

PSL is currently being used by hardware designers, architects, and verification engineers, for specification and verification of hardware. Newcomers to PSL begin by learning the language basics: the syntax and semantics. However, experience shows that after mastering these building blocks, some users still encounter difficulties when they begin writing PSL properties in real-life settings. Beginners, facing the wealth of choices offerred by the language, require guidance in choosing the best way to write a property.

This document aims to alleviate such initial difficulties by guiding the reader through a set of real-life examples, representing typical properties that may be encountered in everyday work. Each example is explained both graphically and verbally, and is accompanied by a discussion, in order to answer questions that frequently arise in the writing of PSL.

The main section of this document, Section 2, presents a set of properties that were used in a real-life hardware verification project. Each property is accompanied by alternative properties, either equivalent to the original property or otherwise related to it. The equivalent properties illustrate various styles of writing, and provide advice on good writing style. Related – but non-equivalent – properties are presented in order to prevent commonly occurring errors, by pointing out subtle differences that an inexperienced user may not be aware of. Common forms of properties are

presented, to help the user identify the template that is most appropriate for each case.

All of the properties are accompanied by timing diagrams, which illustrate possible behaviours described by each property, and point out the differences between non-equivalent properties.

An additional set of typical properties is presented in Section 3. These properties were not taken from a single hardware design project, but were collected from various contexts, in order to illustrate interesting uses of PSL constructs.

Purpose

The purpose of this document is to present a diverse set of PSL properties, demonstrating typical usage of PSL for hardware specification and verification, and accompanied by detailed explanations and discussion.

Intended Audience

This handbook is intended for individuals who use PSL for hardware specification or verification. It is especially aimed at new users of the language, who do not have much practical experience in writing PSL. It is assumed that readers are familiar with the syntax and semantics of PSL, as defined in the PSL Language Reference Manual [1]. Readers are also assumed to be familiar with the basic concepts of hardware design.

Background

PSL is currently being used by hardware designers, architects, and verification engineers, in order to write properties for specification of hardware designs, and for their verification.

For a newcomer who is beginning to use PSL, an important starting point is the official Language Reference Manual (LRM) [1]. The LRM provides the reader with the necessary building blocks for writing properties: the syntax and semantics of the language. However, experience shows that after mastering these building blocks and gaining a good understanding of the language, some users still encounter difficulties when they begin writing PSL properties in real-life settings. Commonly occurring questions are, “what is the best style for the property that I need to write?”, “if these two properties are equivalent, which of them should I choose?”, and also, “why is my property incorrect?”.

This document aims to alleviate such initial difficulties by guiding the reader through a set of real-life examples, representing typical properties that may be encountered in everyday work. Each example is explained both graphically and verbally, and is accompanied by a discussion, in order to answer questions that frequently arise in the writing of PSL.

Contents

Table of Revisions iii
Authors iii
Executive Summary iii
Purpose iv
Intended Audience iv
Background iv
Contents vii
Table of Figures viii
Glossary xi
1 Introduction 1
Technical Notes 2
2 Example – Verification of a Real-life Block 3
Block Description 3
Inputs 4
Outputs 4
Internal Signals 5
Properties for Verifying the Block 5
Control Logic 5
Status Indication 25
Data Consistency 34
Internal Signals 36
Checking the Environment 37
3 General Supplementary Properties 41
Reference to the Past – Prev() 41
Reference to the Past – Endpoint 42
Sampling According to Enabling Signal 43
Sequence Conjunction 44
Named Sequences 45
Restrict 46
Abort 47
Named Properties 49
Assume 50
Overlapping Concatenation of Sequences 51
Acknowledgements 53
4 References 55

Table of Figures

Figure 1 - Interface Signals of the Data Receiver Block 3

Figure 2 - Example trace for property 2 6

Figure 3 - A trace satisfying property 2.1, but contradicting property 2.1.A 6

Figure 4 - Example trace for property 2.2 7

Figure 5 - Example trace for property 2.3 7

Figure 6 - Example trace for property 2.4 8

Figure 7 - Example trace for property 2.4.A 8

Figure 8 - Example trace for property 2.5 9

Figure 9 - Example trace for property 2.6 9

Figure 10 - A trace satisfying property 2.6, but contradicting property 2.6.A 10

Figure 11 - Example trace for property 2.7 10

Figure 12 - A trace satisfying property 2.7.A, but contradicting property 2.7 11

Figure 13 - Example trace for property 2.8 11

Figure 14 - Example trace for property 2.9 12

Figure 15 - Example trace for property 2.10 13

Figure 16 - Example trace for property 2.11 13

Figure 17 - Example trace for property 2.12 14

Figure 18 - A trace satisfying property 2.12, but contradicting property 2.12.A 14

Figure 19 - Example trace for property 2.13 15

Figure 20 - Example trace for property 2.14 15

Figure 21 - Example trace for property 2.15 16

Figure 22 - Example trace for property 2.16 17

Figure 23 - Example trace for property 2.17 17

Figure 24 - Example trace for property 2.18 18

Figure 25 - Example trace for property 2.19 19

Figure 26 - Example trace for property 2.20 20

Figure 27 - A trace satisfying property 2.20.A, but contradicting property 2.20 20

Figure 28 - Example trace for property 2.21 21
Figure 29 - Example trace for property 2.21.A 21
Figure 30 - Example trace for property 2.22 21
Figure 31 - Example trace for property 2.23 22
Figure 32 - Example trace for property 2.24 23
Figure 33 - Example trace for property 2.25 23
Figure 34 - Example trace for property 2.26 24
Figure 35 - Counter-example for property 2.27 25
Figure 36 - Example trace for property 2.28 26
Figure 37 - Example trace for property 2.29 27
Figure 38 - Example trace for property 2.30 27
Figure 39 - Example trace for property 2.31 28
Figure 40 - Example trace for property 2.32 29
Figure 41 - Example trace for property 2.33 30
Figure 42 - Example trace for property 2.34 31
Figure 43 - Example trace for property 2.35 32
Figure 44 - Example trace for property 2.36 33
Figure 45 - Example trace for property 2.37 35
Figure 46 - Example trace for property 2.38 35
Figure 47 - Example trace for property 2.39 36
Figure 48 - Example trace for property 2.40 37
Figure 49 - Example trace for property 2.41 38
Figure 50 - Example trace for property 2.42 38
Figure 51 - Counter-example for property 2.43 39
Figure 52 - Example trace for property 3.1 41
Figure 53 - Example trace for property 3.2 42
Figure 54 - Example trace for property 3.3 43
Figure 55 - Example trace for property 3.4 44
Figure 56 - Example trace for property 3.5 44
Figure 57 - Example trace for property 3.5.A 45
Figure 58 - Example trace for property 3.6 46
Figure 59 - Example trace for property 3.7 46
Figure 60 - Example trace for property 3.7.A 47
Figure 61 - Example trace for property 3.8 47

Figure 62 - A trace satisfying property 3.8.A 48

Figure 63 - Example trace for property 3.9 49

Figure 64 - Example trace for property 3.10 50

Figure 65 - Example trace for property 3.11 50

Figure 66 - A trace satisfying property 3.11.A, but contradicting property 3.11 51

Figure 67 - Example trace for property 3.12 51

Glossary

Behaviour

A succession of states of the design.

Block

A group of interconnected cells. A block may contain instances of other blocks.

Boolean expression

An expression that yields a logical value.

Computation path

A succession of states of the design, such that the design can actually transition from each state on the path to its successor.

Cycle

One iteration of the evaluation process. At an evaluation cycle, the state of the design is recomputed (and may change).

Describes

A property describes the set of behaviours for which the property holds.

Design

A model of a piece of hardware, described in some hardware description language (HDL). A design typically involves a collection of inputs, outputs, state elements, and combinational functions that compute next state and outputs from current state and inputs.

Holds

A term used to talk about the meaning of a property. Loosely speaking, a property holds in the first cycle of a path iff the path exhibits the behaviour described by the property. The definition of holds for each form of property is given in the PSL Language Reference Manual [1].

Property

A collection of logical and temporal relationships between and among subordinate Boolean expressions, sequential expressions, and other properties that in aggregate represent a set of behaviours.

PSL

Property Specification Language, the language for specification of designs upon which PROSYD is based.

Satisfies

A path satisfies a property if the property holds on that path.

Sequence

A SERE that is enclosed in curly braces.

Sequential expression

A SERE.

SERE

A Sugar Extended Regular Expression – a finite series of terms that represent a set of behaviours.

Specification

The process of defining the expected behaviour of a hardware design.

Timing diagram

A diagram that graphically displays a finite computation path.

Trace

A timing diagram.

Verification

The process of falsifying or verifying the functional and performance requirements of a design, be it a chip, board, or system. Many different kinds of verification tools are in use today, including simulation, formal verification, various types of physical analysis tools, emulation, and rapid prototyping. Most design verification strategies employ many or all of these approaches to assure the reliability of the final product prior to its manufacture

Verify

To prove that a property holds on a particular design.

1 Introduction

PSL is a rich and expressive language. It offers a wide variety of constructs, which can be used for writing properties in many different styles.

A newcomer to PSL may face this wealth of choices and ask, “how do I choose the best way to write my property?”. Can SEREs be used? Or ’always’ and ’next’? if both ’until’ and ’before’ are appropriate, which of them would be preferable?

As in any language, beginners often start writing PSL by mimicking existing examples. Many of the properties that a user needs to write have already been written many times before, perhaps with slight variations. By examining the PSL code of experienced users, a beginner may learn important guidelines for writing clearly, elegantly, and correctly.

In Section 2, such an example of PSL code is provided. This section presents a set of properties that were used in a real-life hardware verification project. The section begins with a short description of the hardware block that was verified. Following that, the properties are presented in succession, so that the reader gradually becomes acquainted with the signals of the block and learns their expected behaviour. As the section progresses, new PSL concepts are introduced. Earlier properties use a small set of simple operators, and more complex operators are gradually added, with recommendations for using them correctly. Common forms of properties are presented, to help the user identify the template that is most appropriate for each case.

Many of the properties are accompanied by *equivalent properties*. These are intended to demonstrate alternative ways of writing the same property, with a discussion of the advantages of each form. The alternative properties illustrate various styles of writing, and provide advice on good writing style.

Other properties are accompanied by *related properties*. These properties are intentionally different than the original properties (and are clearly marked as such in the text). Their purpose is to prevent commonly occurring errors, by pointing out subtle differences that an inexperienced user may not be aware of.

All of the properties are accompanied by *timing diagrams*. For each property, a timing diagram is provided in order to illustrate one of the possible behaviours described by the property. Timing diagrams are also used in order to point out the differences between related – but non-equivalent – properties.

In Section 3, an additional set of typical properties is presented, complementing the properties of Section 2. These properties were not taken from a single hardware design project, but were collected from various contexts. Each of them illustrates

an interesting use of PSL constructs. As in Section 2, each property is accompanied by alternative properties (either equivalent or related), and by timing diagrams.

Technical Notes

The properties in this document are written in the Verilog flavour of PSL. They comply with Accellera PSL version 1.1.

Each property is presented with accompanying explanations and figures, including:

- formulation of the property in English
- formulation of the property in PSL
- an example of a trace that satisfies the property, presented as a timing diagram
- a discussion of the structure of the property and the PSL constructs that it uses
- additional properties, which may be equivalent to the original property, or otherwise related to it
- additional trace examples for clarifying the related properties

2 Example – Verification of a Real-life Block

In this section, we describe a real-life example in which PSL properties were used for verifying a hardware block. We provide a basic description of the block itself, and present the properties used in verifying it.

Block Description

The "data receiver" block is responsible for receiving data packets from a "producer" block, and passing them on to a "consumer" block. It prepares the data in the format required by the consumer, and provides the data pacing control.

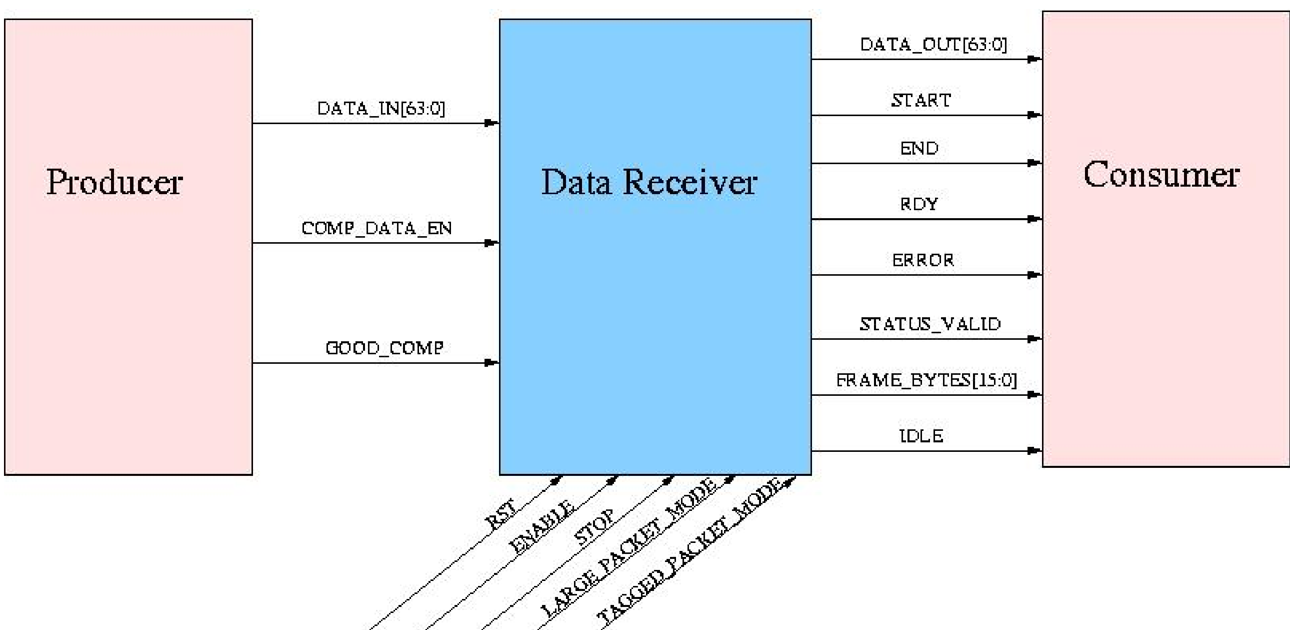

Figure 1: Interface Signals of the Data Receiver Block

The verification process of the data receiver block, on which this example is based, consisted mainly of verifying the interface of the block. Therefore we will describe the interface signals that were verified, without going into the internal details of the block. A few internal signals will be described, since they are mentioned in the properties.

Inputs

- RST – Reset signal
- ENABLE – Indicates that operations are enabled
- STOP – When active, the reception of the current packet should be interrupted
- DATA_IN[63:0] – The input data bus
- LARGE_PACKET_MODE – Indicates that large packets should be supported
- COMP_DATA_EN – Indicates that the result of a data-consistency computation is available
- GOOD_COMP – Indicates that a data-consistency computation was successful
- TAGGED_PACKET_MODE – Indicates that specially tagged packets are supported

Outputs

- DATA_OUT[63:0] – The data/status bus. Bits [31:0] contain data, [63:32] contain the status word
- START – Indicates that the first frame of the received packet is available on the DATA_OUT bus
- END – Indicates that the last frame of the received packet is available on the DATA_OUT bus
- RDY – Indicates that valid information is available on the DATA_OUT bus
- ERROR – Indicates that the received frame is truncated as the result of a detected error
- STATUS_VALID – Indicates that the status of the received frame is available on the DATA_OUT bus
- FRAME_BYTES[15:0] – Presents the size of the last frame received (the number of bytes); contains valid data only when STATUS_VALID is asserted

- IDLE – Indicates idle state – no signals are driven on the interfaces

Internal Signals

- STATE – The state of the internal state machine. Possible values are {data, active1, active2, idle}
- BYTES_COUNTER[15:0] – Counts the number of bytes in the received frame
- SAVED_BYTES_COUNTER[15:0] – An auxiliary counter
- LAST_LENGTH[15:0] – Saves the length of the last frame received

---

Properties for Verifying the Block

The properties presented here are only a representative subset of all the properties used in verifying the block. They are not sufficient in themselves for full verification of the block.

Control Logic

The main part of the verification process consists of verifying the behaviour of the control signals. This section includes properties for checking individual signals, for checking interaction between signals, and for checking complex sequences of events.

Property 2.1 *As long as RST is still up, START, END, and RDY should not be asserted.*

2.1 `assert {RST[+]} |-> {!START && !END && !RDY} ;`

An example trace satisfying property 2.1 is shown in Figure 2.

The property does not start with 'always', therefore it only refers to cases where RST is asserted in the first cycle, and holds continuously for some number of cycles after that.

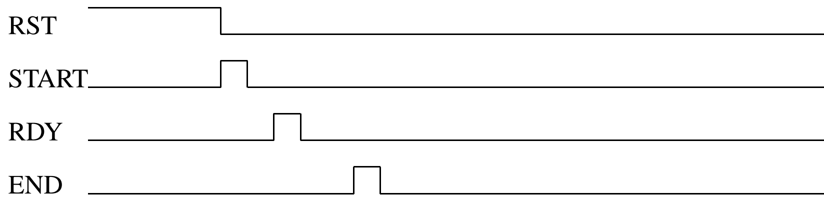

Figure 2: Example trace for property 2

Related properties

2.1.A `assert always {RST} |–> {!START && !END && !RDY} ;`

This property refers to any cycle in which RST is asserted, regardless of whether RST was asserted in previous cycles.

A trace satisfying property 2.1, but contradicting property 2.1.A, is shown in Figure 3.

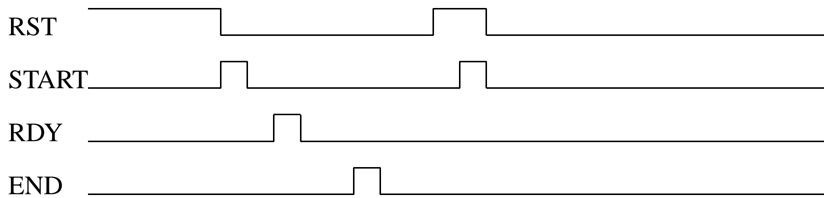

Figure 3: A trace satisfying property 2.1, but contradicting property 2.1.A

Property 2.2 *The signals START, STATUS_VALID, and ERROR are pulses: they can only be asserted for one cycle, and must be deasserted in the next cycle.*

2.2.A `assert always (START –> next (!START)) ;`

2.2.B `assert always (STATUS_VALID –> next (!STATUS_VALID)) ;`

2.2.C `assert always (ERROR –> next (!ERROR)) ;`

An example trace satisfying property 2.2 is shown in Figure 4.

Properties 2.2.A-2.2.C are all of the form 'always (p–> next q)', where p and q are boolean expressions. Properties of this type are very commonly used in PSL.

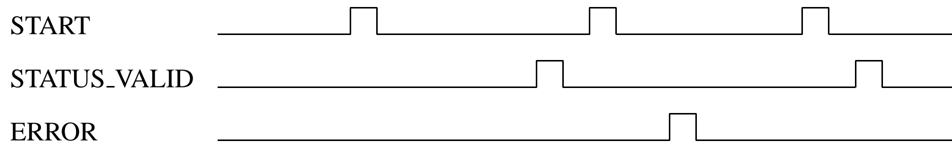

Figure 4: Example trace for property 2.2

Equivalent properties

2.2.D `assert never {ERROR[*2]} ;`

This property is equivalent to property 2.2.C. In contrast to property 2.2.C, which is written in a positive form, property 2.2.D is written in a negative form: it states that a succesion of two 'ERROR' cycles is not allowed.

Property 2.3 *END is a pulse, unless START and RDY are asserted.*

2.3
```
assert
    always (END -> next (!END || (END && START && RDY))) ;
```

An example trace satisfying property 2.3 is shown in Figure 5.

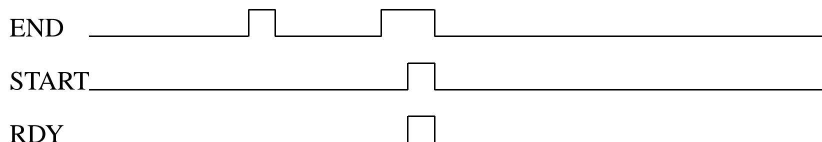

Figure 5: Example trace for property 2.3

This property contains some redundancy, and can be written more concisely without changing the meaning (see equivalent properties below). However, this redundancy improves the readability of the property.

Equivalent properties

2.3.A `assert always (END -> next (!END || (START && RDY))) ;`

2.3.B `assert always (END -> next (END -> (START && RDY))) ;`

Both of these alternatives are equivalent to property 2.3, but their meaning might be less obvious.

Property 2.4 *ERROR and RDY can only be asserted together for one cycle; in the next cycle, at least one of them must be deasserted.*

2.4 `assert always (ERROR && RDY -> next (!ERROR || !RDY)) ;`

An example trace satisfying property 2.4 is shown in Figure 6.

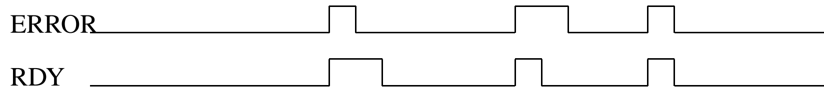

Figure 6: Example trace for property 2.4

This property might be easily confused with one that requires "*... in the next cycle* both *ERROR and RDY must be deasserted*" (see related properties below).

Related properties

2.4.A `assert always (ERROR && RDY -> next (!ERROR && !RDY)) ;`

In contrast to property 2.4, which allows one of the signals ERROR or RDY to remain asserted after the cycle in which both are asserted, property 2.4.A requires that both should be deasserted in the following cycle.

A trace satisfying property 2.4.A is shown in Figure 7.

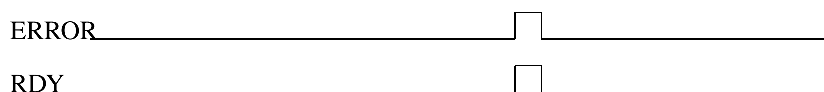

Figure 7: Example trace for property 2.4.A

Property 2.5 *If END is asserted for the first time, then it must have been preceded by a START.*

2.5 `assert {!START[+]} |-> {!END} ;`

An example trace satisfying property 2.5 is shown in Figure 8.

This property does not begin with 'always', therefore it only checks the first time that END is asserted. The property is written in negative form: as long as START is not asserted, END must not be asserted.

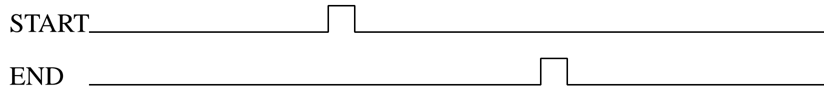

Figure 8: Example trace for property 2.5

Equivalent properties

2.5.A `assert START before END ;`

The ’before’ operator provides a very natural way of expressing this property.

Property 2.6 *RDY must not be asserted before the first START (may be asserted on the same cycle as START).*

2.6 `assert !RDY until START ;`

An example trace satisfying property 2.6 is shown in Figure 9.

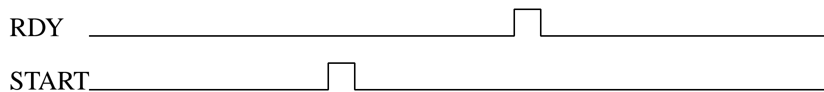

Figure 9: Example trace for property 2.6

This property does not begin with ’always’, therefore it only checks the first time that START is asserted

Related properties

2.6.A `assert !RDY until!  START ;`

The ’until’ operator has several variants. Property 2.6 uses a weak variant, without a ’!’, and therefore the property does not demand that START be asserted eventually. It will be satisfied by a behaviour in which START is never asserted (note that in this case, RDY must also never be asserted). The use of the ’until!’ variant in property 2.6.A means that it will only be satisfied by behaviours in which START is asserted at least once.

A trace satisfying property 2.6, but contradicting property 2.6.A, is shown in Figure 10.

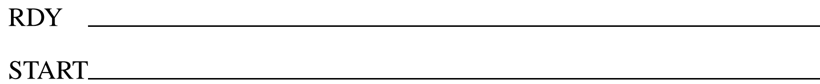

Figure 10: A trace satisfying property 2.6, but contradicting property 2.6.A

Property 2.7 *When START is asserted, RDY must be asserted, and must remain up until one of the signals END, STOP, or ERROR is asserted (including the cycle on which they are asserted).*

2.7
```
assert
    always ( START -> (RDY until_ (END || STOP || ERROR)));
```

An example trace satisfying property 2.7 is shown in Figure 11.

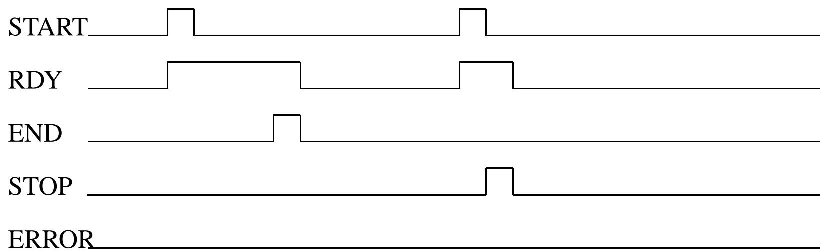

Figure 11: Example trace for property 2.7

The operator 'until_' is an inclusive variant of 'until', and therefore it requires that RDY remain asserted in the cycle where END, STOP, or ERROR is asserted

Related properties

2.7.A
```
assert
    always ( START -> (RDY until (END || STOP || ERROR)));
```

As opposed to property 2.7, this property uses the non-inclusive 'until'.

A trace satisfying property 2.7.A, but contradicting property 2.7, is shown in Figure 12.

Property 2.8 *After one of the signals END, STOP, or ERROR is asserted, RDY must be deasserted, and must stay down until the next START.*

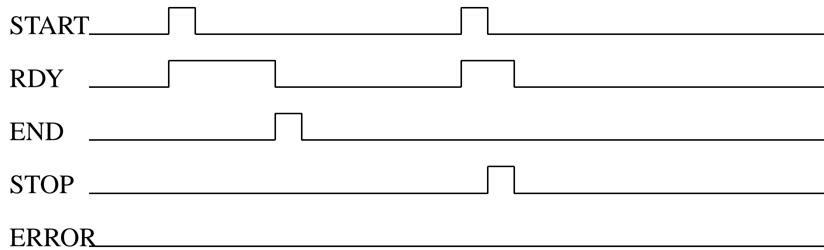

Figure 12: A trace satisfying property 2.7.A, but contradicting property 2.7

2.8
```
assert
    always ((END || STOP || ERROR) -> next (!RDY until START));
```

An example trace satisfying property 2.8 is shown in Figure 13.

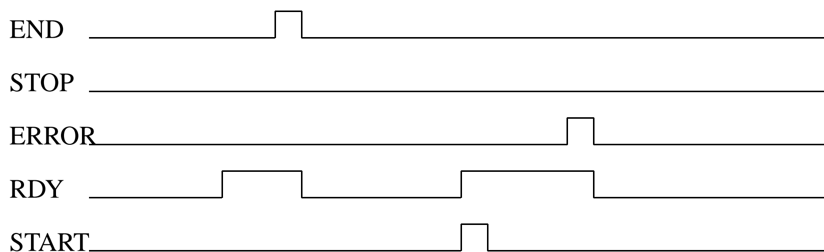

Figure 13: Example trace for property 2.8

This property is a companion to property 2.7; together they describe an "if and only if" requirement: RDY should be asserted if (and only if) we are in an interval starting with a START and ending with END, STOP, or ERROR

Equivalent properties

2.8.A
```
assert
    always {END || STOP || ERROR} |=> {!RDY[*]; START}
```

This equivalent property demonstrates the use of a SERE instead of the 'until' operator.

Property 2.9 *The signals END, STOP, and ERROR may only be asserted if RDY is asserted.*

2.9
```
assert always ((END || STOP || ERROR) -> RDY) ;
```

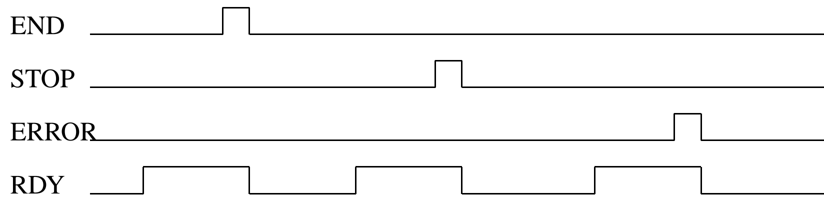

Figure 14: Example trace for property 2.9

An example trace satisfying property 2.9 is shown in Figure 14.

This property is of the form 'always (p)', where p is a boolean expression. Properties of this type are very commonly used in PSL. They are sometimes called *invariants*, because the boolean condition must hold on every cycle.

Related properties

2.9.A `assert always (END -> RDY) ;`

2.9.B `assert always (STOP -> RDY) ;`

2.9.C `assert always (ERROR -> RDY) ;`

Properties 2.9.A, 2.9.B, and 2.9.C, when taken in conjunction, are equivalent to property 2.9. In some cases users may prefer to break down a complex property into several simpler properties. For example, if property 2.9.A fails, the user immediately knows that END has been asserted illegally. In contrast, when property 2.9 fails then the user only knows that one of the signals END, STOP, and ERROR has been asserted illegally, and must check further to discover which of them has caused the illegal behaviour.

Property 2.10 *There exists a transaction that reaches its end – either END or STOP or ERROR.*

2.10 `assert EF (RDY && (END || STOP || ERROR)) ;`

An example trace satisfying property 2.10 is shown in Figure 15.

The 'EF' operator is part of the OBE (Optional Branching Extension) of PSL, which includes CTL properties. The 'E' operators (EF, EG, EX) can be used for existential properties of the form "there exists a computation path in which ...".

Related properties

2.10.A `assert eventually! (RDY && (END || STOP || ERROR)) ;`

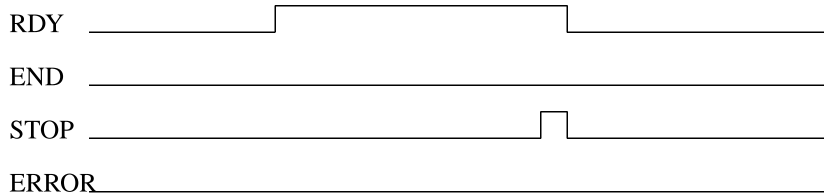

Figure 15: Example trace for property 2.10

This property is very similar to property 2.10. However, the meaning of this property is, "on every computation path, a transaction end is eventually reached (either END or STOP or ERROR)". In contrast, property 2.10 does not make a requirement for *every* computation path. It is satisfied even if there exists only one possible computation path that eventually reaches a transaction end.

The difference between property 2.10 and property 2.10.A cannot be demonstrated by a single timing diagram, since this difference relates to all possible computation paths.

Property 2.11 *If STOP is received while STATUS_VALID is inactive, then on the next cycle RDY, END, and START must be inactive, and STATUS_VALID must be asserted.*

2.11
```
assert {[*]; STOP && !STATUS_VALID} |=>
       {!RDY && !END && !START && STATUS_VALID} ;
```

An example trace satisfying property 2.11 is shown in Figure 16.

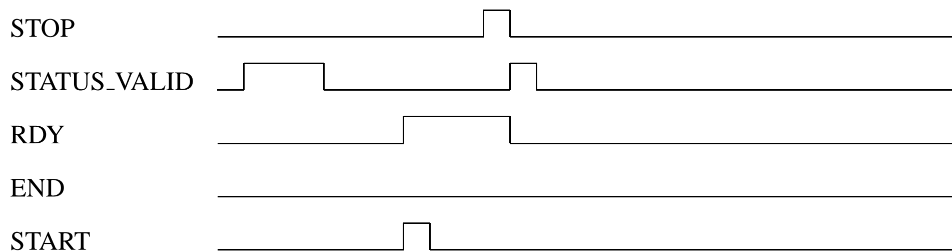

Figure 16: Example trace for property 2.11

This property is of the form '{[*]; p} |=> { q }', where p and q are boolean expressions. This is equivalent to the form 'always (p−> next q)' mentioned in property 2.2, as demonstrated in property 2.11.A below.

Equivalent properties

2.11.A
```
assert
  always ((STOP && !STATUS_VALID) ->
    next (!RDY && !END && !START && STATUS_VALID)) ;
```

Property 2.12 *STATUS_VALID must not rise before the first START.*

2.12 `assert (START before STATUS_VALID) ;`

An example trace satisfying property 2.12 is shown in Figure 17.

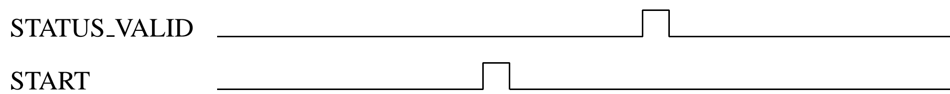

Figure 17: Example trace for property 2.12

This property uses a weak variant of the 'before' operator, without a '!'. Therefore it is satisfied even if START is never asserted (see related properties below).

Related properties

2.12.A `assert (START before!  STATUS_VALID) ;`

This property, as opposed to property 2.12, uses 'before!', which is a strong variant of 'before'. Therefore, it is satisfied only by paths in which START is asserted at some point.

A trace satisfying property 2.12, but contradicting property 2.12.A, is shown in Figure 18.

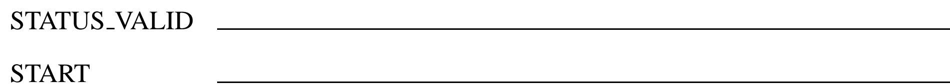

Figure 18: A trace satisfying property 2.12, but contradicting property 2.12.A

Property 2.13 *ERROR must not be asserted together with END or with STATUS_VALID.*

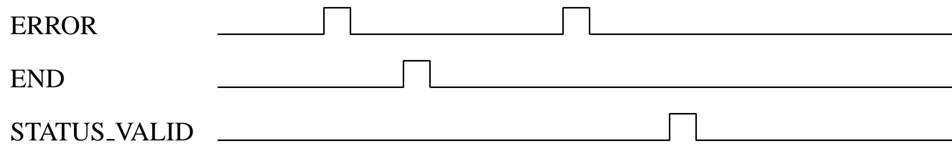

Figure 19: Example trace for property 2.13

2.13 `assert always ( (END || STATUS_VALID) -> !ERROR) ;`

An example trace satisfying property 2.13 is shown in Figure 19.

This is a property of the form 'always (p -> q)', which contains a simple logical implication. Such a property can be written equivalently with the reverse implication: 'always (!q -> !p)'. In some cases, the reverse implication may be more intuitive and thus more readable.

Equivalent properties

2.13.A `assert (ERROR -> (!END && !STATUS_VALID)) ;`

This is equivalent to property 2.13, but uses the reverse implication.

Property 2.14 *ERROR must not be asserted between an END and the following START (from one cycle after the END until one cycle after the START.*

2.14 `assert always (END -> next (START before ERROR)) ;`

An example trace satisfying property 2.14 is shown in Figure 20.

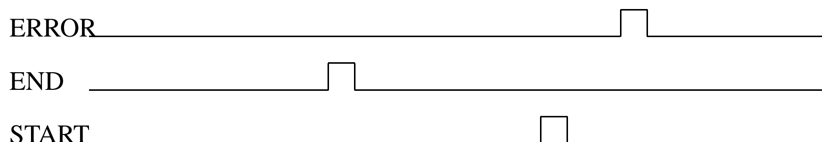

Figure 20: Example trace for property 2.14

This property is formulated as a positive requirement: if END is asserted, then START and ERROR must subsequently behave in a certain manner. An alternative approach is to describe an illegal behaviour, using a sequence, and then forbid it by applying the 'never' operator. This is demonstrated in property 2.14.A below.

Equivalent properties

2.14.A `assert`
```
never {END; {(!START)[*]; START; true} && {ERROR[=1]}};
```

Property 2.15 *STATUS_VALID must not be asserted together with START or with END.*

2.15 `assert never (STATUS_VALID && (START || END)) ;`

An example trace satisfying property 2.15 is shown in Figure 21.

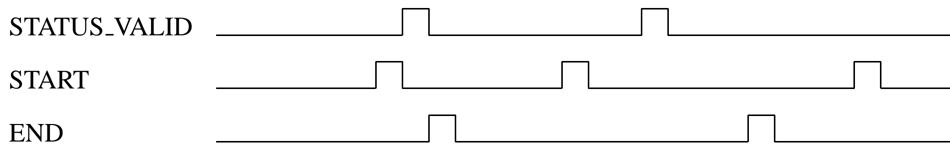

Figure 21: Example trace for property 2.15

We have previously encountered properties of the form 'always (p −> q)', where p and q are boolen expressions. Such a property can be written equivalently as 'never (p && !q)'. Thus property 2.15 is equivalent to property 2.15.A below.

Equivalent properties

2.15.A `assert always (STATUS_VALID −> (!START && !END)) ;`

Property 2.16 *If we are at the end of a transaction (START is down, END is up), and STATUS_VALID is down, then STATUS_VALID must be asserted before the next time that START is asserted.*

2.16
```
assert always (!START && !STATUS_VALID && END −>
  next (STATUS_VALID before START )) ;
```

An example trace satisfying property 2.16 is shown in Figure 22.

The operators 'before' and 'until' are strongly related. For example, a property of the form 'p before q' (with boolean expressions p and q), can be written equivalently as '!q until_ p' (note the use of inclusive 'until_'). Thus property 2.16 is equivalent to property 2.16.A below.

Equivalent properties

2.16.A
```
assert always (!START && !STATUS_VALID && END −>
  next (!START until_ STATUS_VALID)) ;
```

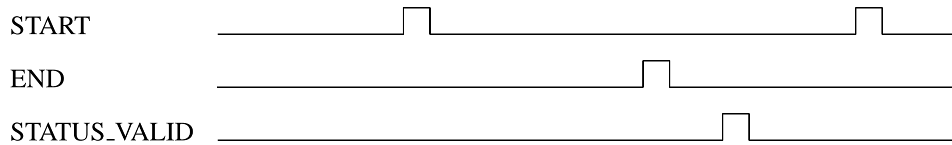

Figure 22: Example trace for property 2.16

Property 2.17 *If START is down, STATUS_VALID may only rise for one cycle before the next time that START is asserted.*

2.17
```
assert always (!START && STATUS_VALID ->
    next (!STATUS_VALID until START)) ;
```

An example trace satisfying property 2.17 is shown in Figure 23.

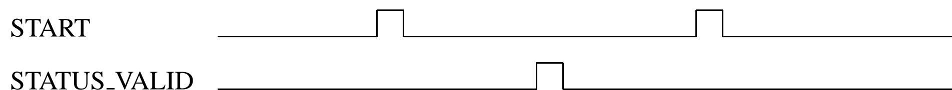

Figure 23: Example trace for property 2.17

The intention of this property is to examine an interval between two assertions of START, and to check that STATUS_VALID rises at most once in such an interval. Property 2.17 uses the 'until' operator to detect the end of the interval (i.e., the next assertion of START).

In property 2.17.A below, the interval between two assertions of START is described more explicitly, using sequence operators.

Equivalent properties

2.17.A
```
assert never { {START[=0]} && {STATUS_VALID[=2] } ;
```

This property describes two behaviours: one in which START is never asserted (i.e., an interval between two assertions of START), and one in which STATUS_VALID is asserted twice (i.e, more than once). The property states that the two behaviours must never happen concurrently.

Property 2.18 *ERROR must not rise if we have seen STATUS_VALID and haven't seen the next START yet (from the cycle after the STATUS_VALID until the cycle of the START).*

2.18 `assert {[*]; STATUS_VALID; !START[*]} |=> {!ERROR} ;`

An example trace satisfying property 2.18 is shown in Figure 24.

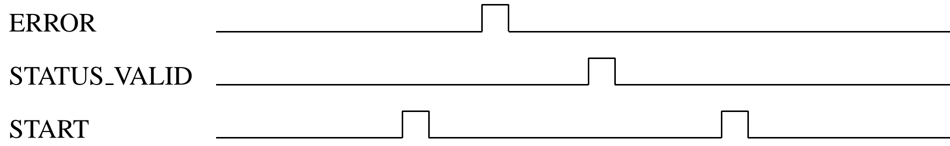

Figure 24: Example trace for property 2.18

In this property, the right-hand-side of the suffix implication requires '!ERROR'. However, it makes no requirement regarding START, so when we reach the cycle in which the right-hand-side is evaluated, START may rise (i.e., we may have reached the next START), but ERROR must still stay deasserted.

Equivalent properties

2.18.A
```
assert never { STATUS_VALID;
        {{START[=0]} | {START[ -> ]}} && {ERROR[=1]}} ;
```

This property is equivalent to property 2.18. It describes an illegal behaviour, and uses 'never' to forbid this behaviour. The illegal behaviour consists of two behaviours occurring concurrently: in the first, START is either not asserted at all, or is asserted on the last cycle of the described behaviour. In the second, ERROR is asserted once. The property states that these two behaviours must never occur together.

2.18.B `assert always (STATUS_VALID -> next(!ERROR until_ START));`

This property is also equivalent to property 2.18, but does not use sequence operators. It uses the 'until_' operator to detect the next assertion of START. The inclusive 'until_' is used because ERROR must remain deasserted on the cycle in which the next assertion of START occurs.

Property 2.19 *The auxiliary signal INSTARTSV indicates that we have seen a START, and haven't seen a STATUS_VALID since then. It rises one cycle after the START, and falls one cycle after the STATUS_VALID.*

2.19
```
endpoint INSTARTSV =
    { START && !STATUS_VALID; (!STATUS_VALID)[*]; true } ;
```

Figure 25: Example trace for property 2.19

An example trace satisfying property 2.19 is shown in Figure 25.

Property 2.19 is an endpoint declaration (not an assertion). It declares a boolean signal named INSTARTSV. The sequence in the endpoint declaration describes a behaviour, and the signal INSTARTSV will be asserted on any cycle that is the final cycle of such a behaviour. Specifically, if START is asserted without STATUS_VALID, then INSTARTSV will be asserted on the following cycle, and will remain asserted as long as !STATUS_VALID holds (this is expressed by (!STATUS_VALID)[*] in the sequence). The extra 'true' cycle at the end of the sequence ensures that when STATUS_VALID is eventually asserted, INSTARTSV will not be deasserted on the same cycle, but only on the following cycle.

Equivalent properties

2.19.A
```
endpoint INSTARTSV =
  { {START} : {(! STATUS_VALID)[+]} ; true } ;
```

This property, which is equivalent to property 2.19, uses the fusion operator ':' to express the same meaning more concisely. The fusion operator concatenates its two sequence operands with one cycle of overlap. In this case, {START} overlaps the first cycle of {(! STATUS_VALID)[+]}.

Property 2.20 *ERROR may be asserted only while INSTARTSV is true, i.e., we are at least one cycle after a START and haven't seen a STATUS_VALID since then (except perhaps in the current cycle)*

2.20
```
assert always (!INSTARTSV -> !ERROR) ;
```

An example trace satisfying property 2.20 is shown in Figure 26.

This property demonstrates the use of the endpoint INSTARTSV, which was defined in property 2.19. The endpoint INSTARTSV functions as a boolean signal, and can be used wherever a boolean signal is allowed.

Related properties

2.20.A
```
assert always {ERROR} |->
        { START && !STATUS_VALID; (!STATUS_VALID)[*]; true } ;
```

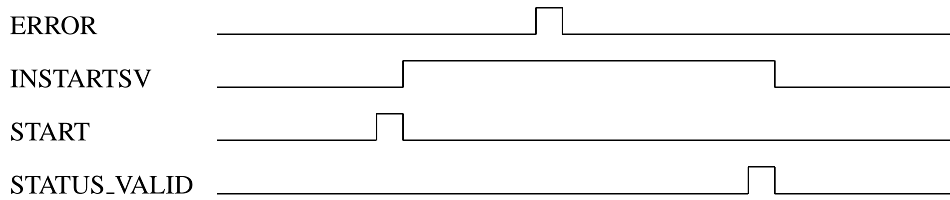

Figure 26: Example trace for property 2.20

This property is not equivalent to property 2.20. We may be misled into thinking so, since 'always (ERROR –> INSTARTSV)' is equivalent to property 2.20. However, replacing an endpoint instance with its defining sequence does not, in general, produce an equivalent property.

A trace satisfying property 2.20.A, but contradicting property 2.20, is shown in Figure 27.

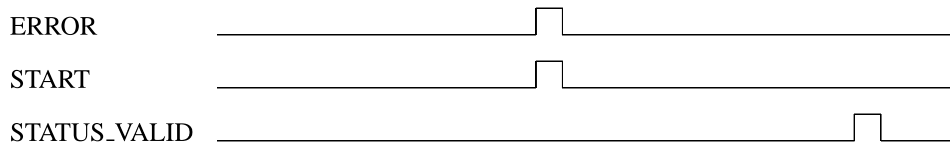

Figure 27: A trace satisfying property 2.20.A, but contradicting property 2.20

Property 2.21 *END may be asserted only during the time frame beginning with START (inclusive) and continuing until STATUS_VALID rises (inclusive).*

2.21 `assert always (END -> (START || INSTARTSV)) ;`

An example trace satisfying property 2.21 is shown in Figure 28.

The definition of INSTARTSV states that it rises one cycle after START is asserted. For property 2.21, we allow END to be asserted earlier, on the same cycle in which START is asserted. For this reason, we must use the condition '(START || INSTARTSV)'. Compare this with property 2.21.A below.

Related properties

2.21.A `assert always (END -> INSTARTSV) ;`

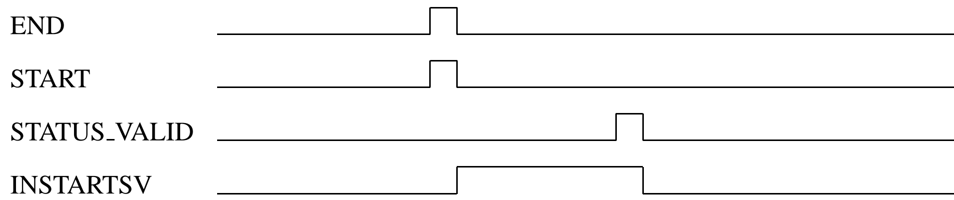

Figure 28: Example trace for property 2.21

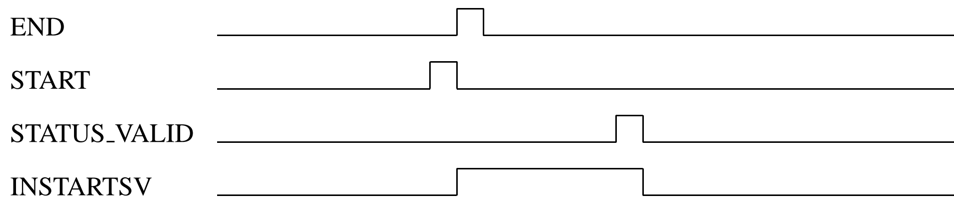

Figure 29: Example trace for property 2.21.A

This property is similar to property 2.21, with the difference that it does not allow END to rise on the same cycle in which START rises, but only one cycle later.

A trace satisfying property 2.21.A is shown in Figure 29.

Property 2.22 *END must not rise if we have seen STATUS_VALID and haven't seen the next START yet (from the cycle after the STATUS_VALID until the cycle before the START).*

2.22 `assert {[*]; STATUS_VALID; !START[*]} |=> {!END || START} ;`

An example trace satisfying property 2.22 is shown in Figure 30.

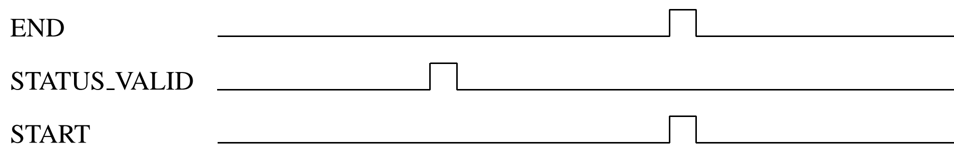

Figure 30: Example trace for property 2.22

Compare this property with property 2.18. That property describes an interval in which ERROR must not be asserted, which is very similar to the interval in which

END must not be asserted. The difference is only in one cycle: ERROR must remain deasserted on the cycle of the next START, while END may rise on that cycle. This difference is reflected in the right-hand-side of the suffix implication.

Equivalent properties

2.22.A `assert never {STATUS_VALID; {START[=0]} && {END[=1]}};`

This property can be compared with property 2.18.A, which describes a larger set of forbidden behaviours.

2.22.B
```
assert
    always (STATUS_VALID -> next (!END until START));
```

This property can be compared with property 2.18.B. The subtle difference is the use of 'until_' vs. 'until'.

Property 2.23 *IDLE is asserted 2 cycles after RST falls.*

2.23 `assert {[*]; fell(RST); [*2] } |-> {IDLE} ;`

An example trace satisfying property 2.23 is shown in Figure 31.

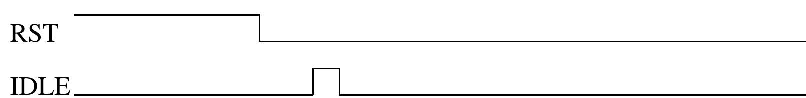

Figure 31: Example trace for property 2.23

This property uses overlapping suffix implication, so the right-hand-side sequence {IDLE} should hold, starting on the last cycle of the left-hand-side sequence {[*]; fell(RST); [*2] }. In other words, if RST falls on cycle n, IDLE should be asserted on cycle $n+2$.

Equivalent properties

2.23.A `assert always (fell(RST) -> next[2](IDLE)) ;`

This property uses 'next[2]', which is an extension of the 'next' operator. If the current cycle is n, then 'next[2]' refers to cycle $n+2$.

Property 2.24 *(While RST is down) IDLE may rise only if ENABLE is down.*

2.24 `assert {[*]; !RST && ENABLE && !IDLE} |=> {!IDLE} ;`

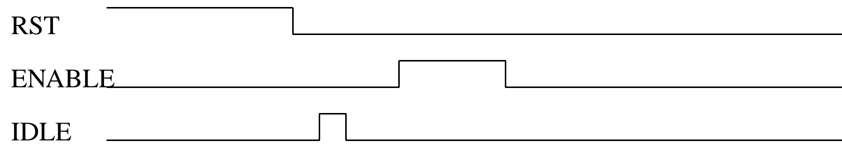

Figure 32: Example trace for property 2.24

An example trace satisfying property 2.24 is shown in Figure 32.

This property describes a situation in which IDLE must not *rise*, i.e. change from 0 to 1. This can be seen clearly in property 2.24.A, in which the built-in function 'rose' is used.

Equivalent properties

2.24.A
```
assert
    always ((!RST && ENABLE) -> next (!rose(IDLE)));
```

Property 2.25 *If START is asserted with ENABLE inactive, and ENABLE was inactive in the previous cycle and remains inactive in the next cycle, then START must be deasserted in the next cycle.*

2.25
```
assert
    {[*]; !ENABLE[+]; START && !ENABLE; !ENABLE} |-> {!START};
```

An example trace satisfying property 2.25 is shown in Figure 33.

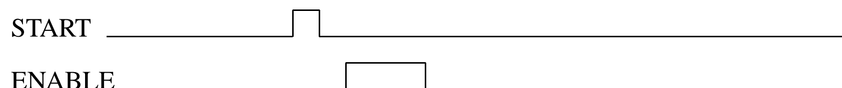

Figure 33: Example trace for property 2.25

The formulation of this property in PSL closely follows the English phrasing. In contrast, the equivalent property 2.25.A below describes the same behaviour, using a different approach. In some cases, a different formulation of a property in PSL provides better insight to the reader about the behaviour that is being described.

Equivalent properties

2.25.A assert

```
{[*];
  {(!ENABLE)[*3:inf]} && {[*] ; true; START; true}}
|-> {!START} ;
```

This property is equivalent to property 2.25. On the left-hand-side, it clearly states that at least 3 cycles are being discussed; ENABLE must be deasserted on all three of them, and START must be asserted in the second of the three.

Property 2.26 *If there are are two occurences of ENABLE rising with STATE=active1, and no START occurs between them, then within 3 cycles of the second rise of ENABLE, START must occur.*

2.26 assert

```
{ [*];
  { rose(ENABLE) && STATE==active1;
  (rose(ENABLE) && STATE==active1)[ -> ]} && {START[=0]}}
|=> {[*0:2] ; START} ;
```

An example trace satisfying property 2.26 is shown in Figure 34.

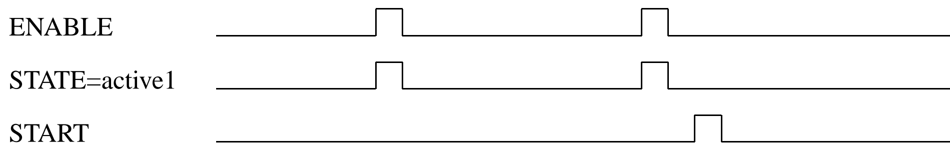

Figure 34: Example trace for property 2.26

On the left-hand-side of this property, a behaviour is described in which ENABLE and STATE receive various values. Throught this behaviour, it is required that START remain deasserted. This is a situation that is often encountered in properties: we need to describe a complex behaviour, and in addition we need to require that a given event – often a 'cancel' or an 'abort' – should not occur throughout the complex behaviour. A convenient mechanism for writing such a property is to use the SERE && operator combined with the [=0] operator. For example, if $S1$ is a SERE describing a complex behaviour, and we need to specify that ABORT is never asserted throughout the occurrence of $S1$, we can write '{$S1$} && {ABORT[=0]}'.

Equivalent properties

2.26.A assert

```
{ [*]; { rose(ENABLE) && STATE==active1;
  !(rose(ENABLE) && STATE==active1)[*];
```

```
rose(ENABLE) && STATE==active1} && {(!START)[*]}}
|=> {[*0:2] ; START} ;
```

Property 2.26 uses the '[->]' operator. Generally, an expression such as '{p[->]}' is equivalent to '{(!p)[*]; p}'. Property 2.26.A uses this equivalent form. In addition, the expression '{START[=0]}' used in property 2.26 is replaced here by the equivalent '{(!START)[*]}'.

Property 2.27 *Show a sequence with 3 transactions (in which END is asserted 3 times).*
Note: *if this assertion fails, then such a sequence exists and can be produced as a counter-example to the assertion.*

2.27 `assert {END[=3]} |-> {false} ;`

A trace that is a counter-example to property 2.27 is shown in Figure 35.

Figure 35: Counter-example for property 2.27

The use of suffix implication with 'false' on the right-hand-side is very convenient for demonstrating that a certain behaviour *does* occur in the computation path. We describe the behaviour on the left-hand-side, and then write the 'false' on the right-hand-side. Any occurrence of the left-hand-side behaviour will cause the assertion to fail, thus proving that an example of the behaviour has actually been found.

Equivalent properties

2.27.A `assert {END[=3]}(false) ;`

Property 2.27.A is equivalent to property 2.27, but uses a different syntax for suffix implication.

Status Indication

The 32 bits of the status word are used for indicating various conditions. The properties in this section check that the bits are indeed activated and deactivated correctly in response to the events that occur.

Property 2.28 *If STOP is received while STATUS_VALID is down, then on the next cycle the STATUS_VALID indication will be asserted. The status word will have "status OK" inactive (DATA_OUT[32]) and "frame reception was stopped" active (DATA_OUT[34]).*

2.28
```
assert {[*]; STOP && !STATUS_VALID; true} |->
    {STATUS_VALID && !DATA_OUT[32] && DATA_OUT[34]} ;
```

An example trace satisfying property 2.28 is shown in Figure 36.

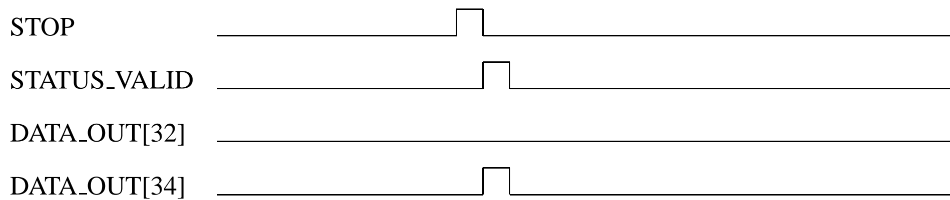

Figure 36: Example trace for property 2.28

The left-hand-side of this property has 'true' as its last element. This is used in order to skip one cycle, regardless of what happens during that cycle. Generally, a property using overlapping suffix implication, of the form '{*S*1; true} |-> {*S*2} (where *S*1 and *S*2 are SEREs), is equivalent to the following property, which uses non-overlapping suffix implication: '{*S*1} |=> {*S*2}'.

Equivalent properties

2.28.A
```
assert always ((STOP && !STATUS_VALID) ->
    next (STATUS_VALID && !DATA_OUT[32] && DATA_OUT[34])) ;
```

This property does not use SEREs. The 'next' operator clearly refers to the next cycle, and there is no need for using 'true'.

Property 2.29 *If the status word is valid (STATUS_VALID is up), the conditions "local link fault" (DATA_OUT[38]) and "remote link fault" (DATA_OUT[39]) must not be active at the same time*

2.29
```
assert never(STATUS_VALID && DATA_OUT[38] && DATA_OUT[39]);
```

An example trace satisfying property 2.29 is shown in Figure 37.

In property 2.29 the signal STATUS_VALID seems to play the same role as all of the other signals in the property – they all appear together in one conjunction.

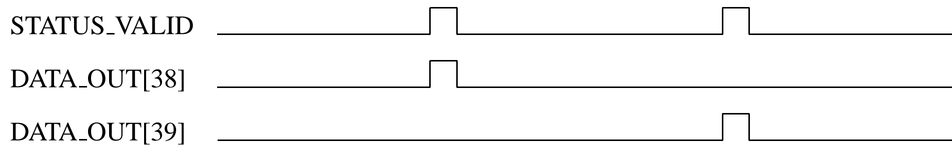

Figure 37: Example trace for property 2.29

However, STATUS_VALID does have a special role in the status indication mechanism of the block: the value of the DATA_OUT bus is only valid when STATUS_VALID is asserted. The equivalent property 2.29.A emphasizes this special role.

Equivalent properties

2.29.A
```
assert
  always (STATUS_VALID -> !(DATA_OUT[38] && DATA_OUT[39]));
```

Property 2.30 *If the status word is valid (STATUS_VALID is up), and one of the conditions "local link fault" (DATA_OUT[38]) and "remote link fault" (DATA_OUT[39]) is active, then the "status ok" condition must be inactive ((DATA_OUT[32]).*

2.30
```
assert always (STATUS_VALID && (DATA_OUT[38] || DATA_OUT[39]) ->
  !DATA_OUT[32]) ;
```

An example trace satisfying property 2.30 is shown in Figure 38.

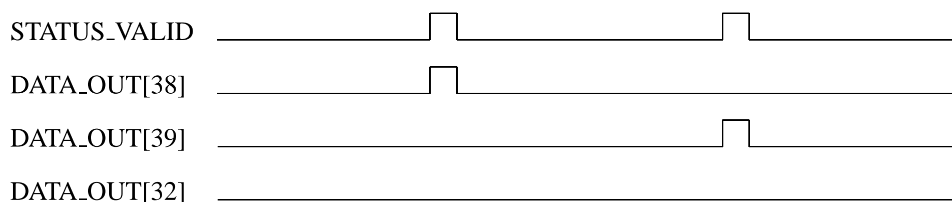

Figure 38: Example trace for property 2.30

This property may seem complex, but it is simply an invariant – a property of the form 'always(p)', with a boolean expression p.

Property 2.31 *If the status word is valid (STATUS_VALID is up), and LARGE_PACKET_MODE is asserted, then a value of FRAME_BYTES[15:0] that exceeds 8000 must cause*

a "frame is too long" indication (DATA_OUT[37]), with "status ok" deactivated (DATA_OUT[32]), unless the "frame reception was stopped" condition is indicated (DATA_OUT[34]).

2.31
```
assert always
(STATUS_VALID && LARGE_PACKET_MODE && FRAME_BYTES[15:0]>8000
-> ((!DATA_OUT[32] && DATA_OUT[37]) || DATA_OUT[34])) ;
```

An example trace satisfying property 2.31 is shown in Figure 39.

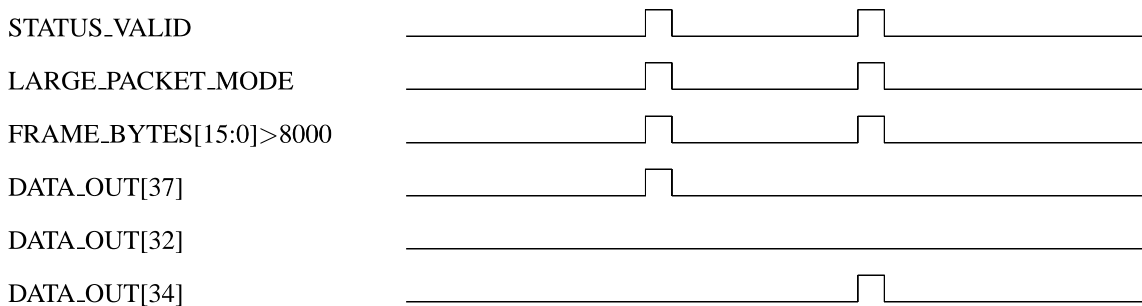

Figure 39: Example trace for property 2.31

Like property 2.30, this property is an invariant with a complex boolean condition. This demonstrates the advantage of specifying properties in PSL, as opposed to an informal specification in English. The complex combination of conjuction, disjunction, and implication is difficult to formulate precisely in English, but has a clear and unambiguous meaning in PSL.

Property 2.32 *If the status word is valid (STATUS_VALID is up), and frame reception was not stopped (DATA_OUT[34] is inactive) and the "frame is too long" condition is not active (DATA_OUT[37]), and one of the "bytes received" bits is active, then the value of FRAME_BYTES[15:0] must be in the range indicated by the active bit:*

- *if "64 bytes recieved" is active (DATA_OUT[44]), then the value of FRAME_BYTES[15:0] must be less than 64*
- *if "64-127 bytes recieved" is active (DATA_OUT[45]), then the value of FRAME_BYTES[15:0] must be more than 63 and less than 128*
- *if "128-maxsize bytes recieved" is active (DATA_OUT[46]), then the value of FRAME_BYTES[15:0] must be more than 127*

2.32.A
```
assert always
(STATUS_VALID && DATA_OUT[44] && !DATA_OUT[34] && !DATA_OUT[37]
-> FRAME_BYTES[15:0] < 64) ;
```

2.32.B
```
assert always
(STATUS_VALID && DATA_OUT[45] && !DATA_OUT[34] && !DATA_OUT[37]
-> FRAME_BYTES[15:0]>63 && FRAME_BYTES[15:0] < 128) ;
```

2.32.C
```
assert always
(STATUS_VALID && DATA_OUT[46] && !DATA_OUT[34] && !DATA_OUT[37]
-> FRAME_BYTES[15:0] >127) ;
```

An example trace satisfying property 2.32 is shown in Figure 40.

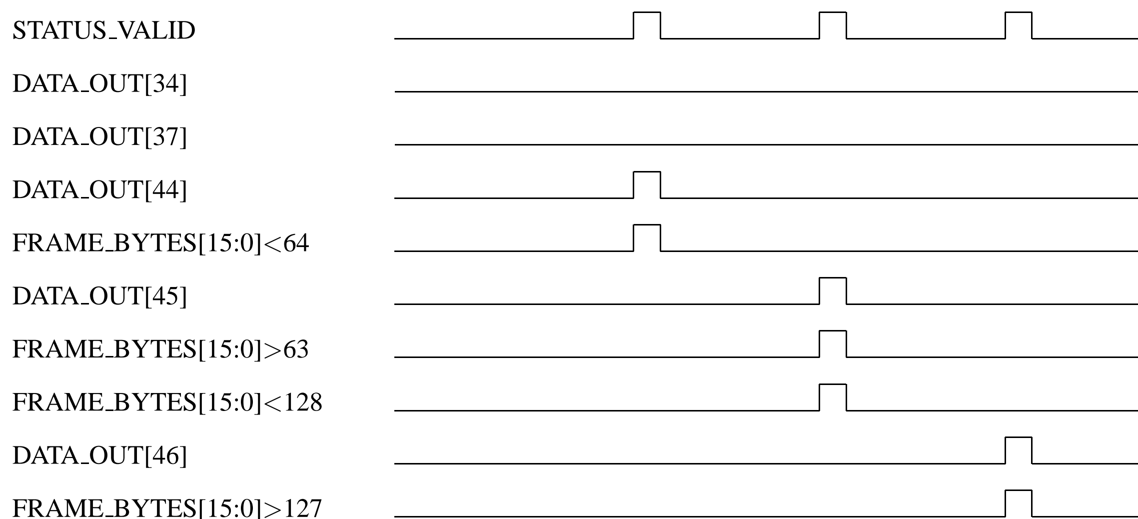

Figure 40: Example trace for property 2.32

The three properties 2.32.A-2.32.C are very similar to each other, and contain some expressions which are repeated in all three. In such a case it may be convenient to define a named property, and use this named property instead of the repeated expression, as demonstrated in properties 2.32.D-2.32.F below.

Equivalent properties

Named property declaration
```
property frame_bytes_is_valid =
    (STATUS_VALID && !DATA_OUT[34] && !DATA_OUT[37]) ;
```

2.32.D
```
assert always
(frame_bytes_is_valid && DATA_OUT[44]
-> FRAME_BYTES[15:0] < 64) ;
```

This property is equivalent to property 2.32.A.

2.32.E
```
assert always
    (frame_bytes_is_valid && DATA_OUT[45]
    -> FRAME_BYTES[15:0]>63 && FRAME_BYTES[15:0] < 128) ;
```

This property is equivalent to property 2.32.B.

2.32.F
```
assert always
    (frame_bytes_is_valid && DATA_OUT[46]
    -> FRAME_BYTES[15:0] >127) ;
```

This property is equivalent to property 2.32.C.

Property 2.33 *If START is asserted with data-consistency result available (COMP_DATA_EN asserted), and on the next cycle COMP_DATA_EN falls and GOOD_COMP is asserted, then the next time that STATUS_VALID is asserted (with no STOP before it), the "bad data-consistency value received" condition (DATA_OUT[33]) must be inactive*

2.33
```
assert
    {[*];
      START && COMP_DATA_EN; !COMP_DATA_EN && GOOD_COMP;
      {STATUS_VALID[ -> ]} && {STOP[=0]; true}}
    |-> {!DATA_OUT[33]} ;
```

An example trace satisfying property 2.33 is shown in Figure 41.

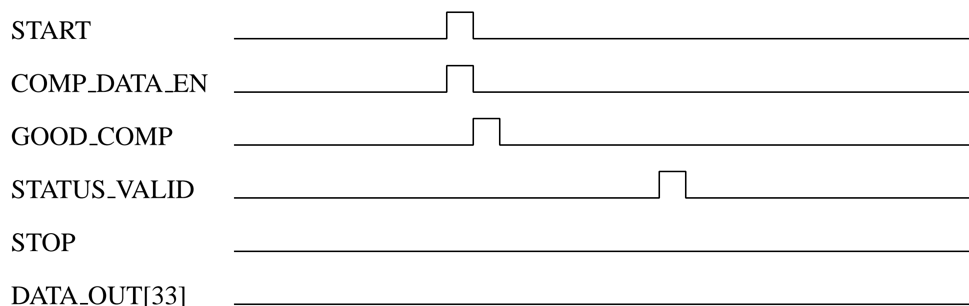

Figure 41: Example trace for property 2.33

The left-hand-side of this suffix implication describes a sequence of events, indicating that a good data-consistency result has been received. The property can be written more clearly by encapsulating these events in a named sequence, and referring to the named sequence in the property.

Equivalent properties

Named sequence declaration `sequence good_comp_reception =`
```
        {START && COMP_DATA_EN; !COMP_DATA_EN && GOOD_COMP} ;
```

2.33.A
```
assert {[*]; {good_comp_reception};
    {STATUS_VALID[ -> ]} && {STOP[=0]; true}}
    |-> {!DATA_OUT[33]} ;
```

This property is equivalent to property 2.33.

Property 2.34 *If START is asserted with data-consistency result available (COMP_DATA_EN asserted), and on the next cycle COMP_DATA_EN falls, but GOOD_COMP is not asserted, then the next time that STATUS_VALID is asserted (with no STOP before it), the "bad data-consistency value received" condition (DATA_OUT[33]) must be active*

2.34
```
assert
    {[*];
      START && COMP_DATA_EN; !COMP_DATA_EN && !GOOD_COMP;
      {STATUS_VALID[ -> ]} && {STOP[=0]; true}}
    |-> {DATA_OUT[33]} ;
```

An example trace satisfying property 2.34 is shown in Figure 42.

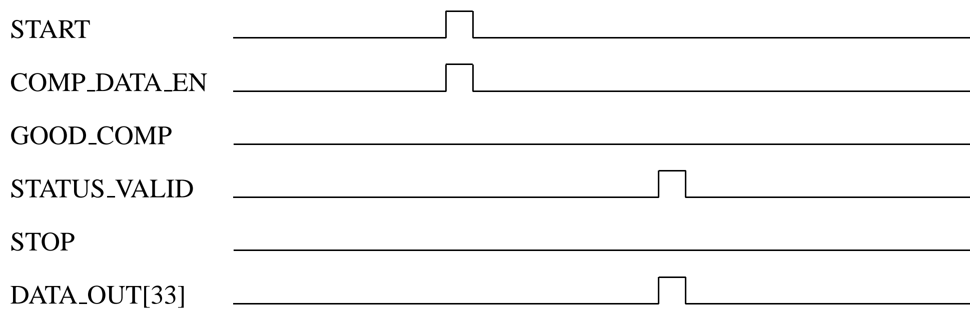

Figure 42: Example trace for property 2.34

This property is the dual of property 2.33. Together, They state that DATA_OUT[33] is asserted if, and only if, a good data-consistency result has been recieved. Property 2.33 can also benefit from the use of a named sequence, as demonstrated below.

Equivalent properties

Named sequence declaration `sequence bad_comp_reception =`

```
        {START && COMP_DATA_EN; !COMP_DATA_EN && !GOOD_COMP} ;
```

2.34.A
```
assert {[*]; {bad_comp_reception};
  {STATUS_VALID[ -> ]} && {STOP[=0]; true}}
  |-> {DATA_OUT[33]} ;
```

This property is equivalent to property 2.34.

Property 2.35 *If BYTES_COUNTER=16 and STATE=data and TAGGED_PACKET_MODE is asserted, then the next time that STATUS_VALID is asserted, the "tagged frame support" condition will be active (DATA_OUT[54]), unless one of the conditions "frame reception was stopped" (DATA_OUT[34]) or "frame is too long" (DATA_OUT[37]) is asserted.*

2.35
```
assert always
  ((BYTES_COUNTER[15:0]==16 && STATE==data
    && TAGGED_PACKET_MODE)
  -> next_event(STATUS_VALID)
    (DATA_OUT[54] || DATA_OUT[34] || DATA_OUT[37] )) ;
```

An example trace satisfying property 2.35 is shown in Figure 43.

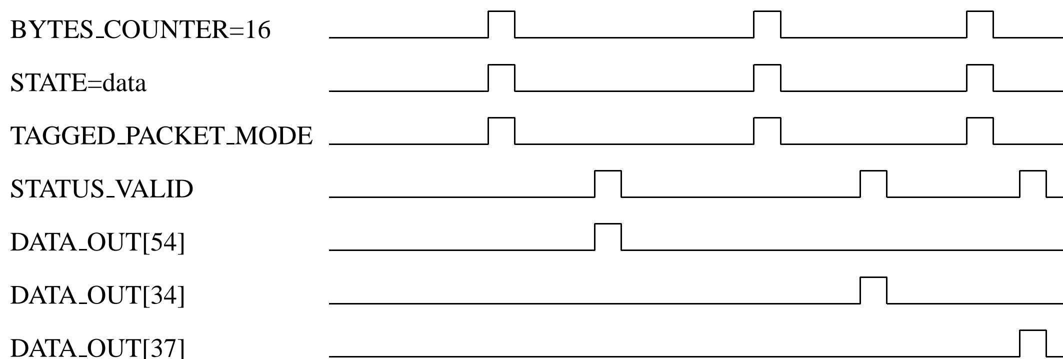

Figure 43: Example trace for property 2.35

This property demonstrates the use of the 'next_event' operator, which is appropriate for expressing conditions such as "the next time that STATUS_VALID is asserted ...". Generally, a property of the form 'p->next_event(q)(r)' (with boolean p, q, and r) can be written equivalently as '{p; q[->]}|->{r}'. This is demonstrated below in property 2.35.A.

Equivalent properties

2.35.A
```
assert always
{BYTES_COUNTER[15:0]==16 && STATE==data && TAGGED_PACKET_MODE;
STATUS_VALID[->]}
|-> {DATA_OUT[54] || DATA_OUT[34] || DATA_OUT[37] } ;
```

Property 2.36 *A bad data indication is characterized by the following conditions: STATE=data, DATA_IN[63] is active, and (DATA_IN[7:0] = RESERVED[7:0]). If we reach a bad data indication and STATUS_VALID is not asserted, then within 3 cycles, STATUS_VALID must be asserted either with an active "illegal code received" condition (DATA_OUT[41]), or with one of the conditions "frame reception was stopped" (DATA_OUT[34]) or "frame is too long" (DATA_OUT[37]) activated.*

2.36
```
assert { [*]; STATE==data && DATA_IN[63] &&
    (DATA_IN[7:0] == RESERVED[7:0]) && !STATUS_VALID;
    !(STATUS_VALID &&
    (DATA_OUT[41] || DATA_OUT[34] || DATA_OUT[37]))[*3]}
  |-> {false} ;
```

An example trace satisfying property 2.36 is shown in Figure 44.

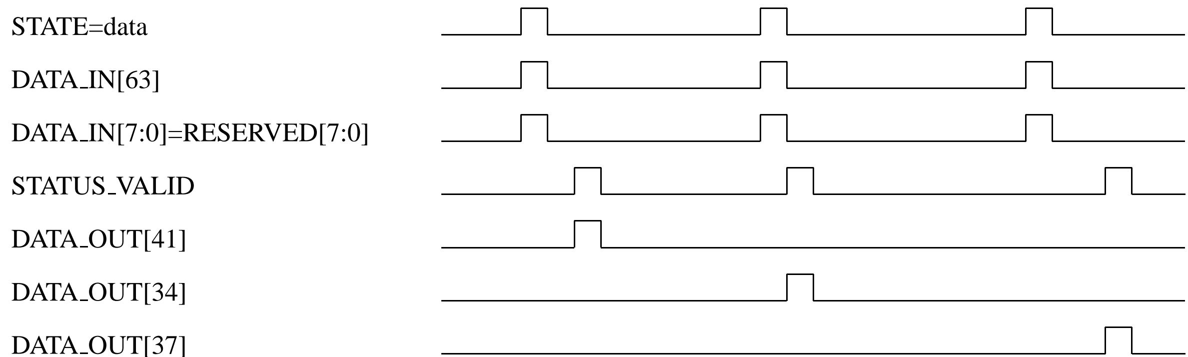

Figure 44: Example trace for property 2.36

This property is written in a negative form: it describes a certain behaviour, and then uses suffix implication with 'false' on the right-hand-side to prohibit this behaviour. The same property can be written in a positive form, as demonstrated below.

Equivalent properties

2.36.A
```
assert { [*]; STATE==data && DATA_IN[63] &&
    (DATA_IN[7:0] == RESERVED[7:0]) && !STATUS_VALID } |=>
    { [*0:2]; STATUS_VALID &&
    (DATA_OUT[41] || DATA_OUT[34] || DATA_OUT[37])} ;
```

This property is equivalent to property 2.36. It states the same requirement positively instead of negatively.

2.36.B
```
assert always
    ((STATE==data && DATA_IN[63] &&
    (DATA_IN[7:0] == RESERVED[7:0]) && !STATUS_VALID) ->
    next_e[1:3](STATUS_VALID &&
    (DATA_OUT[41] || DATA_OUT[34] || DATA_OUT[37]))) ;
```

This property is equivalent to property 2.36.A, but is written without sequence operators. Instead, it uses the operator 'next_e'. An expression of the form 'next_e[1:3](*p*)' means, "*p* must occur at least once within the next 3 cycles".

Data Consistency

The data consistency checks verify that the data received as input later appears correctly on the output bus. They also check the related byte counting mechanism.

Property 2.37 *If STATE=data and BYTES_COUNTER has a current count and DATA_IN contains a current data value, then the next time that SAVED_BYTES_COUNTER reaches that count, DATA_OUT[31:0] will contain that data value.*

2.37
```
assert
    forall counter_val[15:0] in boolean :
      forall data_val[31:0] in boolean :
        always((STATE==data &&
          BYTES_COUNTER[15:0]==counter_val[15:0] &&
          DATA_IN[31:0] == data_val[31:0]) ->
          next_event(SAVED_BYTES_COUNTER[15:0]==counter_val[15:0])
            (DATA_OUT[31:0]==data_val[31:0])) ;
```

An example trace satisfying property 2.37 is shown in Figure 45.

The 'forall' construct is often used for writing properties related to data. It is convenient for writing a property that refers to many possible data values. For example, property 2.37 refers to any possible value of 'data_val[31:0]'. This property demonstrates that 'forall' constructs can be nested, thus checking the cross-product of all possible values of the two 'forall' variables.

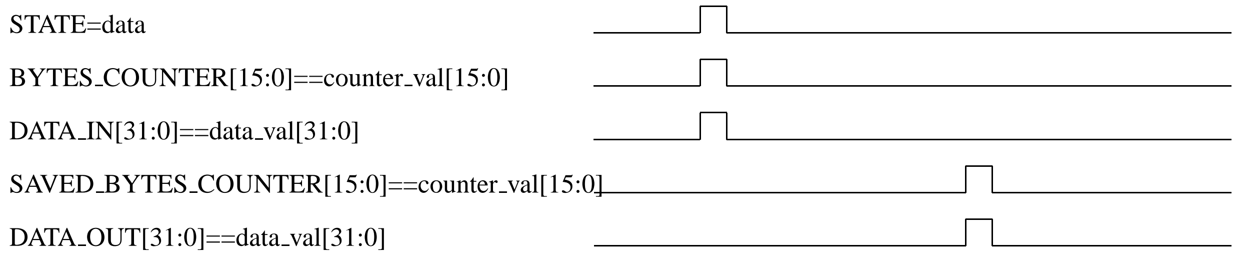

Figure 45: Example trace for property 2.37

Property 2.38 *If STATE=active2 and BYTES_COUNTER has a current value, then the next time that STATUS_VALID is asserted, FRAME_BYTES will have the same value that BYTES_COUNTER now has, unless one of the conditions "frame reception was stopped" (DATA_OUT[34]) or "frame is too long" (DATA_OUT[37]) is asserted.*

2.38
```
assert
  forall val[15:0] in boolean :
    always (STATE==active2 &&
      BYTES_COUNTER[15:0]==val[15:0]) ->
      next_event(STATUS_VALID)
        (FRAME_BYTES[15:0]==val[15:0]
          || DATA_OUT[34] || DATA_OUT[37]);
```

An example trace satisfying property 2.38 is shown in Figure 46.

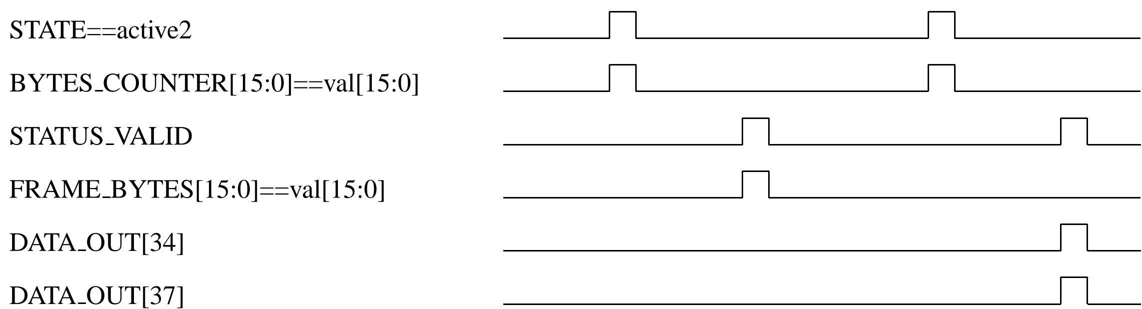

Figure 46: Example trace for property 2.38

This property demonstrates a typical usage of 'forall', in conjunction with 'next_event'. The variable 'val[0:15]' is used for "remembering" the value of BYTES_COUNTER[15:0],

so that this value can be compared to the value of FRAME_BYTES[15:0] the next time that STATUS_VALID is asserted.

Equivalent properties

2.38.A
```
assert
  forall val[15:0] in boolean :
    always
      {STATE==active2 && BYTES_COUNTER[15:0]==val[15:0]);
        STATUS_VALID[->]}
      |-> {FRAME_BYTES[15:0]==val[15:0]
          || DATA_OUT[34] || DATA_OUT[37]};
```

This property is equivalent to property 2.38, but is written with sequence operators instead of 'next_event'.

Internal Signals

Although the verification effort was concentrated mainly on the interface of the block, a few properties were written in order to check critical behaviour of internal signals.

Property 2.39 *If the state machine reaches STATE=active1, it will eventually reach STATE=active2.*

2.39
```
assert
  always(STATE==active1 -> eventually!  (STATE==active2));
fairness STATE!=data;
```

An example trace satisfying property 2.39 is shown in Figure 47.

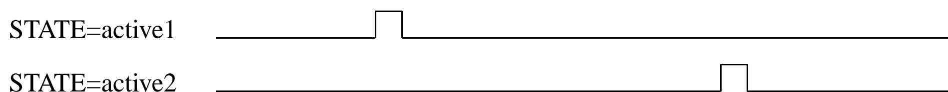

Figure 47: Example trace for property 2.39

Even if our state machine is correct, certain behaviours created by the neighboring blocks might lead it to remain in STATE=data forever. In these behaviours, our property will not hold – we will not eventually reach STATE=active2. However, we are only interested in checking that the property holds on "fair" paths, which do not include an infinite loop at STATE=data. In order to filter out the unwanted paths, we add a fairness statement.

Property 2.40 *If the status word is valid (STATUS_VALID is up), and "frame length out of range" is active (DATA_OUT[36]), and frame reception was not stopped (DATA_OUT[34] is inactive) and the "frame is too long" condition is not active (DATA_OUT[37]), then the value of last_length[15:0] must be greater than 2000.*

2.40
```
assert always
    (STATUS_VALID && DATA_OUT[36]
    && !DATA_OUT[34] && !DATA_OUT[37]
    -> last_length[15:0] > 2000) ;
```

An example trace satisfying property 2.40 is shown in Figure 48.

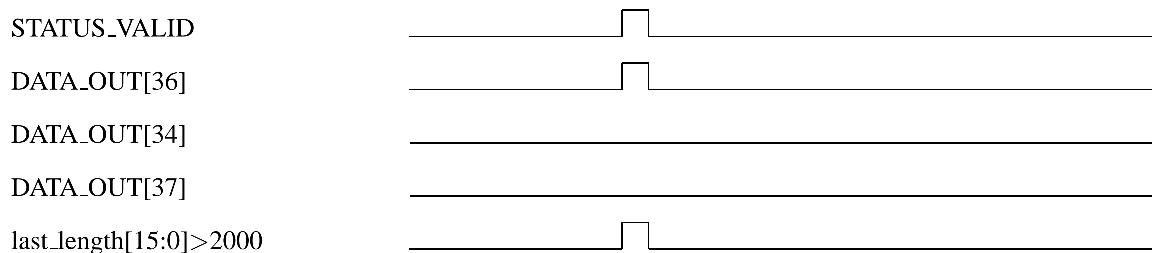

Figure 48: Example trace for property 2.40

This is an additional example of a property that is conceptually simple (an invariant), but is difficult to understand because of the many conditions it involves. This may be improved by using a named property, as demonstrated below.

Equivalent properties

Named property declaration
```
property long_frame_received =
    STATUS_VALID && DATA_OUT[36] && !DATA_OUT[34] && !DATA_OUT[37];
```

2.40.A
```
assert always
    (long_frame_received -> last_length[15:0] > 2000) ;
```

This property is equivalent to property 2.40, but uses a named property to improve readability.

Checking the Environment

The testing environment provides the inputs to the block, simulating the behaviour of its neighboring blocks. As part of the verification procedure, some properties

were written to check that the environment was “alive”, i.e., that certain signals were eventually asserted.

Property 2.41 *There exists a computation path in which ENABLE is eventually asserted, after RST has fallen.*

2.41 `assert EF ( !RST && ENABLE ) ;`

An example trace satisfying property 2.41 is shown in Figure 49.

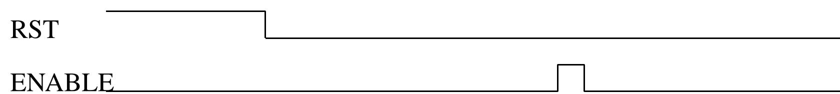

Figure 49: Example trace for property 2.41

The EF operator is typically used for checking that the environment is “alive”. Properties such as this may be written for all of the signals driven by the environment, to ensure that each of them is eventually asserted on some computation path.

Property 2.42 *There exists a computation path in which ENABLE is eventually asserted without IDLE indication, after RST has fallen.*

2.42 `assert EF ( !RST && ENABLE && !IDLE ) ;`

An example trace satisfying property 2.42 is shown in Figure 50.

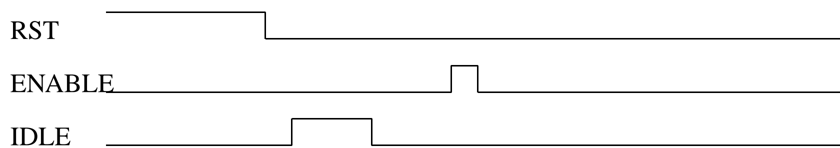

Figure 50: Example trace for property 2.42

Note the subtle difference between this property and property 2.42.A below.

Related properties

2.42.A `assert always (fell(RST) -> EF (ENABLE && !IDLE)) ;`

Part of the requirement of property 2.42 is that RST should eventually be deasserted, on some computation path. So property 2.42 will fail if, for some reason, RST is constantly equal to 1 on all computation paths. Property 2.42.A, in contrast, will not fail in such a case; it only makes a requirement for computation paths in which RST indeed falls at some point.

Property 2.43 *Show a sequence in which ENABLE is asserted (after RST has fallen), and later IDLE becomes inactive.*
Note: *If this assertion fails, then such a sequence exists and can be produced as a counter-example to the assertion.*

2.43 `assert { [*]; !RST && ENABLE; [*]; !IDLE } |-> {false} ;`

A trace that is a counter-example to property 2.43 is shown in Figure 51.

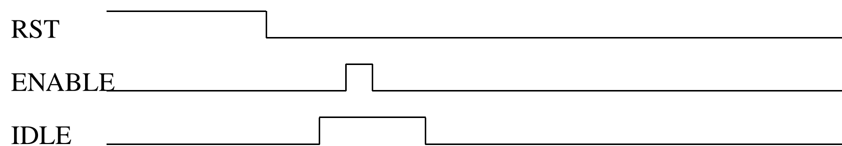

Figure 51: Counter-example for property 2.43

As we have previously seen, we may intentionally write a property stating that a certain behaviour never occurs, for the purpose of seeing the property fail and receiving a counter-example.

Equivalent properties

2.43.A `assert always (!RST && ENABLE -> next always IDLE) ;`

This property is equivalent to property 2.43. It states that after ENABLE is asserted, IDLE remains constantly asserted. If this property fails, then a sequence in which ENABLE is asserted, and IDLE is later deasserted, does indeed exist.

3 General Supplementary Properties

This section presents some additional properties, that are not a part of the case study. These properties complement those in the case study, demonstrating interesting uses of PSL constructs, and some non-obvious solutions.

Reference to the Past – Prev()

Property 3.1 *If ACK is active, then REQ was active 3 cycles ago.*

3.1 `assert always (ACK -> prev(REQ,3));`

An example trace satisfying property 3.1 is shown in Figure 52.

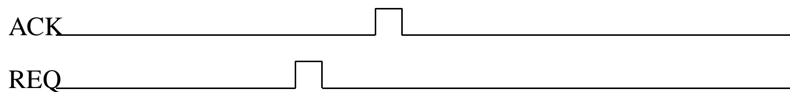

Figure 52: Example trace for property 3.1

This property demonstrates the use of the built-in function 'prev()' in order to refer to previous values of an expression. For example, if 'prev(REQ,3)' is evaluated at cycle n, it returns the value of signal REQ at cycle $n-3$. The function 'prev()' can also be called without the second parameter. In this case, it returns the value in the previous cycle.

Equivalent properties

3.1.A `assert always (ACK -> prev(prev(prev(REQ)))) ;`

This property is equivalent to property 3.1. It uses 'prev()' with a single parameter, which is applied three times, in order to refer to the value of REQ three cycles earlier.

Reference to the Past – Endpoint

Property 3.2 *If ACK is active, then REQ was active somewhere in the past.*

Endpoint declaration `endpoint REQ_END = {REQ; [+]};`

3.2 `assert always (ACK -> REQ_END);`

An example trace satisfying property 3.2 is shown in Figure 53.

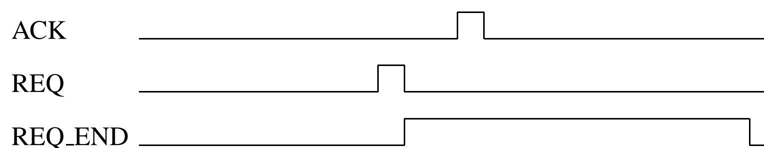

Figure 53: Example trace for property 3.2

PSL does not have temporal operators that refer to the past. For example, there is no operator that is equivalent to 'eventually!', to enable us to require that a certain event happened "sometime in the past". However, this property demonstrates that a similar effect can be acheived by using an endpoint. The endpoint REQ_END rises one cycle after REQ is asserted, and remains asserted constantly from that point onward. Therefore, the condition '(ACK -> REQ_END)' means, "if ACK is asserted now then REQ should have been asserted in the past".

Property 3.3 *If there is a READ, followed by 3 consecutive WRITEs, then in each of the 3 cycles the data written (DO) is equal to the data read (DI).*

Endpoint declaration
```
endpoint DOUT_END(const d) =
    {(DO==d)[*3]};
```

3.3
```
assert forall d in {0:7}:
    {{READ && DI==d}; !WRITE[*]; WRITE[*3]} |-> {DOUT_END(d)};
```

An example trace satisfying property 3.3 is shown in Figure 54.

In this property, an endpoint is used in order to describe two behaviours occurring concurrently. The endpoint DOUT_END(d) is used in order to detect the first behaviour, which consists of three consecutive cycles satisfying 'DO==d'.

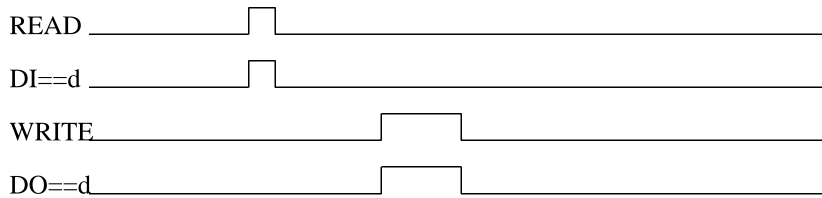

Figure 54: Example trace for property 3.3

Whenever such a sequence is encountered, the endpoint is asserted on the third cycle. The second behaviour is described by the left-hand-side of the suffix implication. The property states that whenever a sequence matching the left-hand-side reaches its end, the endpoint DOUT_END(d) should be asserted. This means that the left-hand-side sequence ends at the same time that the '{(DO==d)[*3]}' sequence ends.

Equivalent properties

3.3.A
```
assert
  forall d in {0:7}:
    never {{{READ && DI==d}; !WRITE[*]; WRITE[*3]}
          && {[*]; (DO!=d)[*1..3]}};
```

This property is equivalent to property 3.3. Instead of using an endpoint to describe concurrent behaviours, it uses the '&&' sequence operator.

Sampling According to Enabling Signal

Property 3.4 *Always, if on 3 consecutive VALIDs, REQ appears, then on the next VALID cycle, ACK holds.*

3.4
```
assert ({[*]; REQ[*3]} |=> {ACK}) @ VALID ;
```

An example trace satisfying property 3.4 is shown in Figure 55.

This property uses the clocking operator '@' in order to single out the cycles in which VALID is asserted.

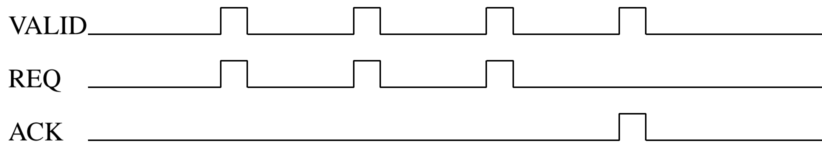

Figure 55: Example trace for property 3.4

Equivalent properties

3.4.A `assert {[*]; (REQ && VALID)[=3]; VALID[->]} |-> {ACK} ;`

This property is equivalent to property 3.4, but does not use the '@' operator. Instead, it explicitly adds the VALID condition to each element of the left-hand-side sequence. The '@' operator helps to write a more concise and readable property.

Sequence Conjunction

Property 3.5 *After a trace with 3 READs and 2 WRITEs that come in any order, READY is eventually active.*

3.5
```
assert {[*]; {READ[=3]} && {WRITE[=2]}} |=>
    {(!READ && !WRITE)[*]; READY}!  ;
```

An example trace satisfying property 3.5 is shown in Figure 56.

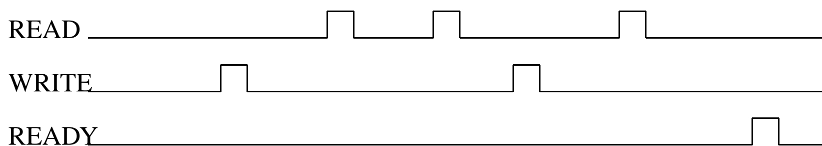

Figure 56: Example trace for property 3.5

This property uses {READ[=3]} to describe a sequence in which READ is asserted 3 times (perhaps not consecutively), while {WRITE[=2]} describes a sequence with 2 WRITEs, which may not be consecutive. The conjunction of the two sequences, using '&&', describes a sequence with 3 READs and 2 WRITEs,

without specifying the order in which the READs and WRITEs should occur. The two sequences must begin and end on the same cycle.

Related properties

3.5.A `assert {[*]; {READ[*3]} & {WRITE[*2]}}|=> {(!READ && !WRITE)[*]; READY}! ;`

This property is not equivalent to property 3.5. It uses [*3] instead of [=3], and [*2] instead of [=2]. Thus, the two sequences that it describes are one in which READ is asserted for three consecutive cycles, and one in which WRITE is asserted for 2 consecutive cycles The two sequences are of different lengths, therefore their conjunction should use the non-length-matching sequence conjunction operator '&'. This conjunction requires that the two sequences start on the same cycle, but they may end on different cycles. Thus, it describes a sequence in which READ and WRITE are both asserted on the first two cycles, and only READ is asserted on the third cycle.

A trace satisfying property 3.5.A is shown in Figure 57.

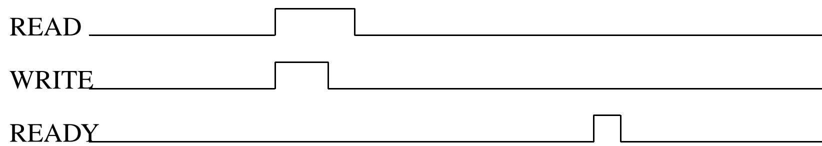

Figure 57: Example trace for property 3.5.A

Named Sequences

Property 3.6 *After a trace with 3 READs and 2 WRITEs that come in any order, READY is eventually active.*

Named sequence declaration `sequence DATA_TRANS(const k, n) = {{READ[=k]} && {WRITE[=n]}};`

3.6 `assert`

```
{[*]; DATA_TRANS(3,2)} |=>
{(!READ && !WRITE)[*]; READY}! ;
```

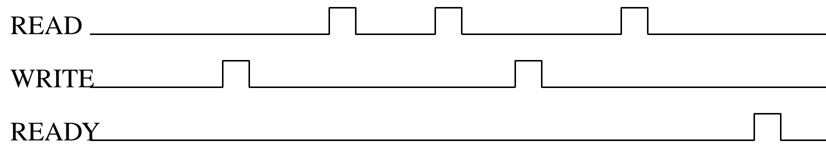

Figure 58: Example trace for property 3.6

An example trace satisfying property 3.6 is shown in Figure 58.

This property demonstrates the use of named sequences with parameters. The sequence DATA_TRANS is defined generically, with parameters k and n. It can then be instantiated with any specific pair of numbers.

Equivalent properties

3.6.A `assert {[*]; {READ[=3]} && {WRITE[=2]} } |=>`
`{(!READ && !WRITE)[*]; READY }! ;`

This property is equivalent to property 3.6, without using a named sequence.

Restrict

Property 3.7 *The RST signal, given by the environment, should be active for 5 cycles in the beginning of every run, and then be inactive forever.*

3.7 `restrict {RST[*5]; !RST[+]} ;`

An example trace satisfying property 3.7 is shown in Figure 59.

Figure 59: Example trace for property 3.7

The 'restrict' directive can be used in order to restrict the execution to certain behaviours. Only computation paths whose prefix matches the given sequence are considered, and all other paths are ignored. The sequence should end with an

unbound sequence operator such as '[*]' or '[+]', to signify that the rest of the computation path, after the matching sequence, may be infinite.

Related properties

3.7.A `restrict {RST[*5]; !RST} ;`

This statement is not equivalent to the one in property 3.7. The sequence does not end with an unbounded sequence operator, and therefore it only describes sequences of length 6. As a result, only computation paths of length 6 will be considered in the model, and any longer path will be ignored.

The following trace satisfies property 3.7.A.

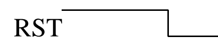

Figure 60: Example trace for property 3.7.A

Abort

Property 3.8 *Every REQ must eventually be acknowledged by ACK, unless INTERRUPT appears.*

3.8 `assert always ((REQ -> F (ACK)) abort INTERRUPT);`

An example trace satisfying property 3.8 is shown in Figure 61.

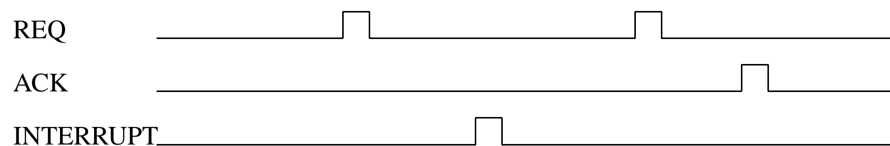

Figure 61: Example trace for property 3.8

The 'abort' operator can be used in order to "cancel future obligations". In a property such as '(REQ -> F (ACK)', if REQ is asserted then ACK should eventually

be asserted. However, the addition of 'abort INTERRUPT' qualifies this requirement, so a behaviour in which REQ is asserted and later INTERRUPT is asserted, but ACK is never asserted, still satisfies the property. In this property the 'abort' operator is applied to a sub-property, and therefore only cancels obligations of the sub-property. In other words, if REQ rises then an assertion of INTERRUPT means that it does not matter whether that REQ is matched by a subsequent ACK. However, if REQ later rises again, after the INTERRUPT, then it creates a new obligation, and so this new REQ must be acknowledged by an ACK (unless another INTERRUPT intervenes). This behaviour is dictated by the fact that the 'abort' is nested inside an 'always' operator.

Related properties

3.8.A `assert (always (REQ -> F (ACK))) abort INTERRUPT;`

This property is not equivalent to property 3.8. In this property, the 'abort' is applied to the entire property, including the 'always', and not only to the inner sub-property. In this property, the assertion of INTERRUPT cancels all future obligations: even if REQ is asserted several times after the assertion of INTERRUPT, it no longer matters whether ACK is asserted afterwards – the property will be satisfied in any case.

A trace satisfying property 3.8.A, but contradicting property 3.8, is shown in Figure 62.

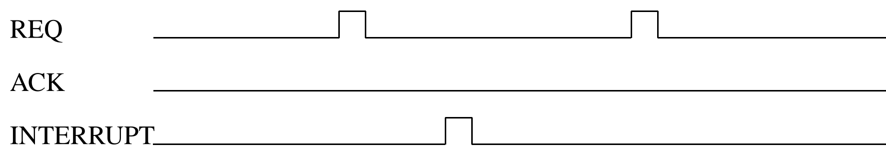

Figure 62: A trace satisfying property 3.8.A

Property 3.9 *If no RETRY occurs during a transaction, which starts with TRANS_START, VALID should be active one cycle after the transaction ends (signal TRANS_END is active).*

3.9 assert always
{TRANS_START; !TRANS_END[*]; TRANS_END} abort RETRY
|-> {true; VALID};

An example trace satisfying property 3.9 is shown in Figure 63.

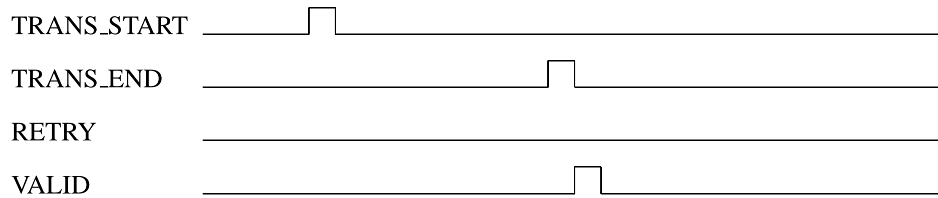

Figure 63: Example trace for property 3.9

In this property the 'abort RETRY' is applied to the left-hand-side of the suffix implication. This describes a behaviour which matches the left-hand-side sequence, but during which RETRY must not be asserted.

Equivalent properties

3.9.A
```
assert always
  {{TRANS_START; !TRANS_END[*]; TRANS_END} && {RETRY[=0]}}
  |-> {true; VALID};
```

This property is equivalent to property 3.9, using sequence conjunction instead of 'abort'.

Named Properties

Property 3.10 *RESULT occurs n cycles after START*

Named property declaration
```
property ResultAfterN (boolean START;
                       const n; property RESULT) =
    always (START -> next[n] (RESULT));
```

3.10
```
assert ResultAfterN(WRITE_REQ, 3, (eventually!  ACK));
```

An example trace satisfying property 3.10 is shown in Figure 64.

This property demonstrates the use of named properties with parameters. The third parameter is of type 'property', and therefore its matching argument in the instantiation can be a property such as '(eventually! ACK)'.

Equivalent properties

3.10.A
```
assert always (WRITE_REQ -> next[3] (eventually!  ACK));
```

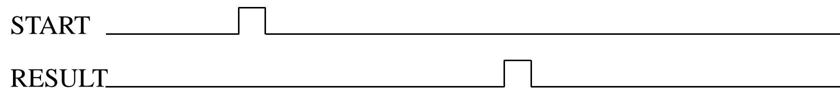

Figure 64: Example trace for property 3.10

This property is equivalent to property 3.10, without using a named property.

Assume

Property 3.11 *READ and WRITE are environment signals, which can be given at any time, but should never be given together.*
(Note that this statement concerns the environment signals, and not the design under test.)

3.11 `assume always (!  (READ && WRITE));`

An example trace satisfying property 3.11 is shown in Figure 65.

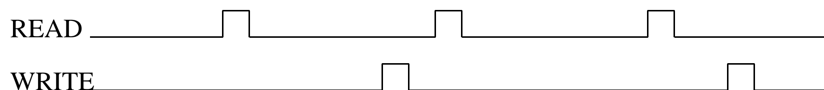

Figure 65: Example trace for property 3.11

The 'assume' directive can be used in order to restrict the execution to certain behaviours. Only computation paths satisfying the given property are considered, and all other paths are ignored In this example the property begins with 'always', in order to make a requirement for all cycles of the path. If the 'always' were omitted, the property would only make a requirement for the first cycle, as demonstrated in property 3.11.A below.

Related properties

3.11.A `assume (!  (READ && WRITE));`

This statement is not equivalent to the one in property 3.11. It only requires that READ and WRITE should not be asserted together on the first cycle.

A trace satisfying property 3.11.A, but contradicting property 3.11, is shown in Figure 66.

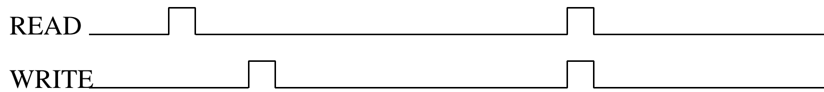

Figure 66: A trace satisfying property 3.11.A, but contradicting property 3.11

Overlapping Concatenation of Sequences

Property 3.12 *If REQ is active, it must be deactivated one cycle after ACK arrives. Note that ACK may already be active when REQ is asserted.*

3.12 `assert {[*]; {REQ}: {!ACK[*]; ACK}} |=> {!REQ} ;`

An example trace satisfying property 3.12 is shown in Figure 67.

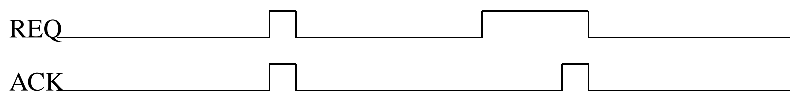

Figure 67: Example trace for property 3.12

In this property, the fusion operator ':' is used in order to concisely describe two sequences with one cycle of overlap between them: the last cycle of the left-hand-side sequence coincides with the first cycle of the right-hand-side sequence.

Equivalent properties

3.12.A
```
assert
    {[*]; {REQ & !ACK; !ACK[*]; ACK} | {REQ & ACK};
    |=> {!REQ} ;
```

This property is equivalent to property 3.12, without using the ':' operator. It separates the sequence '{REQ}:{!ACK[*];ACK}' into two possible behaviours: in the first behaviour there are several cycles in which ACK is down, followed by a cycle in which ACK rises, while in the second behaviour ACK is asserted on the first cycle. In both behaviours, REQ is asserted on the first cycle.

Acknowledgements

We would like to thank the members of the Formal Verification and Testing Technologies Department at the IBM Haifa Research Laboratory for contributing property examples to this document. Special thanks go to Gil Shapir for sharing his verification experience with us.

4 References

[1] *Accelera Property Specification Language: Reference Manual — Version 1.1*
http://www.eda.org/vfv/docs/psl_lrm-1.1.pdf