YAML ATP Input Specification

Eyck Jentzsch

Contents

1 Preface

1.1 About this Specification

This specification describes the specification of traffic profiles to be consumed by a traffic profile unut (TPU). Features implemented by the TPU are derived from AMBA Adaptive Traffic Profiles Specification and further enhanced to allow the generation of specific transaction according to the AXITLM or CHITLM specification.

1.2 References

This manual focuses on the extension of TLM2.0 to model the AMBA AXI and ACE protocol at loosly and approximately timed accuracy. For more details on the protocol and semantics, see the following manuals and specifications:

- AMBA® Adaptive Traffic Profiles Specification, 15 March 2019, ARM IHI 0082A
- AMBA® AXI™ and ACE™ Protocol Specification, 31 March 2020, ARM IHI 0022H

2 Traffic profile definition

Traffic profiles are specified in YAML format. YAML is a well-specified and easily readable markup language. Using this format, we can make use of a pre-verified library, which provides a matured parser. The parser allows to give error messages with references into the input file (line and column). It also offers the possibility to easily integrate further TPU parameters without fiddling in the parser. It simplifies the extension of the traffic profile for farther features.

2.1 YAML file format

Whitespace indentation is used for denoting structure; however, tab characters are not allowed as part of that indentation. Comments begin with the number sign (#), can start anywhere on a line and continue until the end of the line. Comments must be separated from other tokens by whitespace characters [15] If # characters appear inside of a string, then they are number sign (#) literals. List members are denoted by a leading hyphen (-) with one member per line. A list can also be specified by enclosing text in square brackets ([...]) with each entry separated by a comma. An associative array entry is represented using colon space in the form key: value with one entry per line. YAML requires the colon be followed by a space so that scalar values such as http://www.wikipedia.org can generally be represented without needing to be enclosed in quotes. A question mark can be used in front of a key, in the form "?key: value" to allow the key to contain leading dashes, square brackets, etc., without quotes. An associative array can also be specified by text enclosed in curly braces ({...}), with keys separated from values by colon and the entries separated by commas (spaces are not required to retain compatibility with JSON). Strings (one type of scalar in YAML) are ordinarily unquoted, but may be enclosed in double-quotes ("), or single-quotes ('). Within double-quotes, special characters may be represented with C-style escape sequences starting with a backslash (). According to the documentation the only octal escape supported is \0. Full documentation for YAML can be found on its official site. Outlined below are some simple concepts that are important to understand when starting to use YAML.

- Scalars, or variables, are defined using a colon and a space. The entire set of (key value) pairs at the same indentation level (block format) form an associative array (dictionary)

```
integer: 25
string: "25"
float: 25.0
boolean: Yes
```

- Lists can be defined using a conventional block format or an inline format that is similar to JSON.

```
--- # Shopping List in Block Format
- milk
- eggs
- juice

--- # Shopping List in Inline Format
[milk, eggs, juice]

--- # Shopping List as associative array
milk: 1 litre
eggs: 6 pieces
juice: 3 bottles
```

- Strings can also be denoted with a | character, which preserves newlines, or a > character, which folds newlines.

```
data: |
   Each of these
   Newlines
   Will be broken up

data: >
   This text is
   wrapped and will
   be formed into
   a single paragraph
```

2.2 YPRF file format

A YAML profile file (YPRF) forms a parallel context. This means if a list of profiles as specified they are executed in parallel. Subsequent *profile_list* definitions may open sequential or parallel contexts.

All integer values listed in the following sections may be specified in decimal, octal, and hexadecimal notation according to the YAML specification.

2.2.1 Section `profile_list`

A *profile-list* consist of a list of key-value pairs where the following keys are supported:

- parallel_execution - boolean, default: true

  if set to false the profiles in this list are executed sequentailly

- profile - dictionary

  a profile definition

- delay - integer

  number of cycles to wait at this point (only if parallel_execution is set to false)

- post - string

  posts a messgage to the syncronizer

- wait - dictionary

  wait for a posted message of one ore more other TPUs

- message - string

  create a message print at the simulation output

- include - string

  include the content of the denoted file at this point of the file

- profile_list - list

  a nested profile list.

2.2.2 Section `wait`

The wait key allows to wait for events from other TPUs. The wait dictionary can contain 2 keys: inst and event

Syntax	Comment
- wait:	the wait key
inst: <name>	inst name is a regular expresssion allowing to filter the source of the posted message
event: <name>	event name is a regular expression to match the posted message

2.2.3 Section `profile`

The profile keyword is used to start a profile. It must be the first keyword used in a file. When the profile keyword is seen, it sets all parameters back to the default values.

Syntax	Comment
- profile: <name>	If more than one profile defined in the file, each traffic profile must start with a dash and space ("- "). It is recommended to use dash and space even if only one profile is defined. All statements of a profile needs to be indented to the column of the profile key (2 space in this example).
type: <type list>	required, The type designates the type of transaction in the profile. It may be a single type or a inline formatted list. Allowed entry a described below.
count: <int>	Specifies the maximum number of transactions that are generated for the profile. Once the count of transactions issued reaches MAX-TRANS, then the TPU stops generating new transactions for that profile.
generator:	Set the value of a generator parameter, overriding any default value specified in the Traffic Profile Spec (see section 1.2) Table 4-1.
trans_id:	Specifies the way transaction ids are generated
address:	Specifies the address generation scheme
data: random	optional, Specifies how data is generated for each transaction. Since only random is being supported it can be omitted.
timing:	Override the default timings specified in Traffic Profile Spec (see section 1.2) Table 3-1
signals:	Set the value of a protocol signal, overriding the default value specified in the AMBA spec (see section 1.2).

The following types of transactions are supported:

- READ
- WRITE
- ReadNoSnp
- ReadOnce
- ReadOnceCleanInvalid
- ReadOnceMakeInvalid
- ReadClean

- ReadNotSharedDirty
- ReadShared
- ReadUnique
- WriteNoSnpFull
- WriteUniqueFull
- WriteLineUniqueFull
- WriteBackFull
- WriteClean
- WriteEvict
- Evict
- CleanShared
- CleanInvalid
- CleanSharedPersist
- MakeInvalid
- CleanUnique
- MakeUnique
- StashOnceUnique
- StashOnceShared
- WriteUniqueFullStash
- WriteUniquePtlStash

Those types map to the respective AMBA ACE snoop types.

2.2.4 Section `generator`

The generator keyword is used to set the value of a generator parameter, overriding any default value specified in the Traffic Profile Spec (see section 1.2) Table 4-1. The supported generator commands are:

Syntax	Comment
TxnLimit: <limit>	optional, maximum number of outstanding transactions for the profile, default = 1
TxnSize: <bytes>	optional, number of bytes in each transaction request, must be a power of 2, default = 64
Rate: <rate spec>	optional, specify the rate in SI Units (e.g. GBps). If specified all transactions specified using `type:` must be either read or write
Full: <int>	optional, size of the read or write fifo
Start: <level>	optional, one of `empty` or `full`, defines the start level of read or write fifo. If ommitted the default is full for read and empty for write accesses

2.2.5 Section `trans_id`

The trans_id keyword specifies the pattern of ids to use. In AXI, the corresponds to AXID. The form of the command is

Syntax	Comment
type: cycle|unique	defines the kind of generation the transaction id
range: [<lower>,<upper>]	*lower* and *upper* are the bounds on the trans_id.

For type=*cycle*, the id used for each transaction will increment starting with the *lower* to the *upper* and starting over. For type=*unique*, the id used will come from a pool of ids, but an id may only be used for one outstanding transaction at a time.

2.2.6 Section `address`

The address keyword is used to specify the pattern of addresses used in transactions. The simple forms of the command are:

Syntax	Comment
type: <stepping>	Stepping = sequential or random
range: [<base>,<range>]	the starting address and range of generated addresses. base + range-1 = the highest address
alignment: <alignment size>	If omitted the accesses are not aligned. A value of 0 indicates that accesses shall be align to the data bus width

For sequential, the address will increment by the size of the transaction until it issues a transaction that includes highest address, at which point it starts at Base again. For random, the address is randomly selected between Base and Range-1. For random, the starting seed will be fixed. Note that protocol restrictions must be obeyed. As specific examples in AXI:

- the transaction address of burst type INCR must be aligned to the size of the transaction
- the transaction address of burst type WRAP must be aligned to the size of the data bus.

For more complex stepping, there is a two-dimensional command of the form:

Syntax	Comment
type: <stepping>	stepping = sequential or random
range: [<base>,<range>]	the starting address and range of generated addresses. base + range-1 = the highest address
alignment: <alignment size>	If omitted the accesses are not aligned. A value of 0 indicates that accesses shall be align to the data bus width
xrange: <xrange>	
stride: <stride>	

2.2.7 Section `timing`

The timing keyword is used to override the default timings specified in Traffic Profile Spec (see section 1.2) Table 3-1.

Syntax	Comment
<timing name>: <value>	optional, *timing name* denotes a timing name according to the ATP spec, *value* may be a single type or a inline formatted list denoting the number of cycles

The following timing names are supported:

- ARTV

- ARR
- RIV
- RBV
- RBR
- RLA
- AWTV
- AWV
- AWR
- WIV
- WBR
- WBV
- BV
- BR
- BA
- ACTV
- ACR
- CRV
- CRR
- CDIV
- CDBR
- CDBV

2.2.8 Section `signals`

The signal keyword is used to set the value of a protocol signals, overriding the default value specified in AMBA Spec (see section 1.2).

Syntax	Comment
<signal name>: <value>	optional, *signal name* denotes a signal name according to the AMBA spec (see section 1.2), *value* may be a single value or a inline formatted list denoting the value of the signal.

The following signal names are supported:

- AxADDR,
- AxBURST,
- AxCACHE,
- AxID,
- AxLEN,
- AxLOCK,
- AxPROT,
- AxQOS,
- AxREGION,
- AxSIZE,
- BRESP,
- RDATA,
- RRESP,
- WDATA,
- WSTRB,

- AWATOP,
- AWSTASHNID,
- AWSTASHLPID,

Those signals map to the respective AMBA AXI/ACE channel signals.

3 Appendix

3.1 Examples

3.1.1 single profile

```yaml
- profile: "readnosnoop"
  generator:
    TxnLimit: 64
  count: 1500
  type: [ReadNoSnoop]
  address:
    type: sequential
    range: [0x0, 0x20000]
  trans_id:
    type: cycle
    range: [0, 64]
  signals:
    AXCACHE: 0xf
```

3.1.2 Profile with advanced address stepping and rate limitation

```yaml
- profile: "readnosnoop"
  generator:
    TxnLimit: 64
    Rate: 20 GBps
    Full: 1024
  count: 10000
  type: ReadOnce
  address:
    type: twodim
    range: [0x0, 0x80000]
    xrange: 0x40
    stride: 0x100
  trans_id:
    type: cycle
    range: [0, 64]
  signals:
    AXCACHE: 15
```

3.1.3 Sequential profile with syncronization

```yaml
- profile_list:
  - parallel_execution: false
```

```yaml
- message: "start_miss"
- profile_list:
  - profile: "read_once_0x0"
    generator:
      TxnLimit: 64
    count: 1500
    type: [ReadOnce]
    address:
      type: sequential
      range: [0x0, 0x20000]
    trans_id:
      type: cycle
      range: [0, 64]
    signals:
      AXCACHE: 15
- message: "end_miss"
- post: "checkpoint1"
- wait:
    inst: "\\.caiu"
    event: "checkpoint1"
- delay: 1000
- message: "start_hit"
- profile_list:
  - profile: "read_once_0x0"
    generator:
      TxnLimit: 64
    count: 1500
    type: [ReadOnce]
    address:
      type: sequential
      range: [0x0, 0x20000]
    trans_id:
      type: cycle
      range: [0, 64]
    signals:
      AXCACHE: 15
- message: "end_hit"
```