Refinement Map for ILA vs. Verilog Verification

Hongce Zhang

6th January 2019

WARNING: This is a draft specification. As the development of the ILAng tool, we will try to keep compatible with this specification.

Contents

1 The Structure of Refinement Map

The refinement map or refinement relation consist of two parts: *variable mapping* and *instruction completion conditions*. These two parts are specified in two separate files, one referred as `var-map` and the other `inst-cond`.

Other than the two main parts, there are other auxiliary information needed. They are:

- **module naming**: The names of the ILA module and the Verilog module.
- **global invariants**: Some properties that are globally true for the Verilog design that will be checked separately and can be safely assumed when verifying individual instructions.
- **interface signal information**: What does the interface of the Verilog top module look like and what do these signals mean to the tool.
- **uninterpreted function mapping**: What an uninterpreted function inside the ILA model corresponds to.

The structure of the two files is shown in Figure 1 and Figure 2. We will introduce them in detail in the later sections.

2 Module Naming

The module naming section comes first in the `var-map` JSON file. It is a dictionary (map) with two elements. One element should have the key "ILA" and the value of it is what will be used as the instance name of the ILA module. The other element should have the key "VERILOG" with the value to be the instance name of the Verilog module. These names are used in all the expressions later when you want to refer to a variable in Verilog or ILA.

3 Variable Mapping

The variable mapping in the JSON file is a dictionary data structure. The keys are the state variable names in the ILA model while the values are the Verilog variables. There are cases that one Verilog state variable can be mapped to multiple Verilog state variables and the mapping may be special function. So the allowed value field of this JSON dictionary can be:

- A Verilog variable name as a string. If the Verilog variable is in the top module of the design, you can omit the module name (it does not hurt to add it). Otherwise, you must specify the complete hierarchy. For example, if you want to refer to a signal `S` in module `A` that is instantiated with name `IA` in the top module, while the top module's instance name is `IT`, then you should use `IT.IA.S` to refer to it.
- A predicate that has at least a "==" in it.

- A condition-map pair that is in the form of a list or map. If it is given as a list, the first element is regarded as the condition and the second element is regarded as the mapping. It conveys the meaning of "when the condition is true, the ILA and Verilog variables should have the mapping". If the condition is not true, there is no mapping guaranteed. If the condition-map pair is given as a dictionary, it must have two elements. The condition element should have the key "cond" and the map element should have the key "map". The map part could be a string of Verilog variable name or a Verilog expression that works as a predicate on the states of ILA and Verilog.
- A list of condition-value pairs.
- A memory directive, a string of "**MEM**name", where the "name" part should be replaced by the name of the memory state in ILA.

```
{
  "models": {
    "ILA" : <module-name:string> ,
    "VERILOG": <module-name:string> },
  "state mapping": {
    <ILA-variable-name:string> : <Verilog-signal:string>,
    ...
    },
  "interface mapping": {
    <Verilog-signal:string> : <ILA-variable-name:string>,
    ...
    },
  "mapping control": [
    <Verilog-expression:string>,
    ...
    ],
  "functions":{
    <function-name:string>:
      [
        [
          <Verilog-expression:string>, <Verilog-signal:string>,
          <Verilog-expression:string>, <Verilog-signal:string>,
          ...
        ]
      ]
  }
}
```

Figure 1: Structure of Variable Mapping Specification

4 Instruction Completion Condition

The instruction completion condition is specified per instruction. In the JSON file, it is a list of dictionaries. The list does not need to be a full list of instructions. Those not in the list will not be verified. Each dictionary must have an element whose key is "instruction" and the value of this element is the name of the instruction in the ILA model. Besides this name element, it must contain one of the "ready bound" or the "ready signal" element. The "ready bound" specifies a bound that the instruction takes. It is used for instructions that take a fixed number of cycles. Alternatively, one can provide a signal (or a predicate) in the "ready signal" field.

There are several optional fields. They are "start condition", "max bound", "flush constraint", "pre flush-end" and "post flush-end".

The "start condition" field, if provided, should be a list of string. Each string is a Verilog expression that acts as a predicate on the Verilog design. It can be used to constrain the Verilog implementation state, because usually there are more microarchitectural (implementation states) in the design. For the starting state of an instruction, these microarchitectural states should be "consistent" with the visible states in the ILA, and you can use this field to enforce the "consistency" at your discretion.

The "max bound" can be used when "ready signal" field is provided. It provides an assumption that the condition that the "ready signal" field specified will occur in the given number of cycles, and this will limit the model checking to that bound. Usually this is used for the additional environment assumptions about the input (for example, how many cycles at most there are for a request to be served with a response).

The "flush constraint", "pre flush-end" and "post flush-end" signals are used when using the "flushing" verification setting. For this verification setting, you

```
{
  "instructions":
  [
    {
      "instruction"      :<instr-name:string>,
      "ready bound"      :<bound:integer>,
      "ready signal"     :<Verilog-signal:string>,
      "max bound"        :<bound:integer>,
      "start condition"  :<Verilog-expr:string>,
      "flush constraint":<Verilog-expr:string>,
      "pre-flush end"    :<Verilog-expr:string>,
      "post-flush end"   :<Verilog-expr:string>
    }
  ]
}
```

Figure 2: Structure of Instruction Completion Specification

can refer to our paper on the ILA-based verification or the Burch-Dill's approach on processor verification.

5 Global Invariants

In the verification of instructions, we do not assume the design starts from the initial states. This helps us to get a better guarantee of the instruction correctness when only bounded model checking is used. However, if there is no constraints on the starting state of a instruction, there might be spurious bugs just because the design starts from a state that it will never reach when started from the reset state. In order to avoid this false positive, we use global invariants to constrain on the starting state. These invariants help rule out some unreachable states and the tool will generate a separate target to check whether the provided invariants are globally true or not. These invariants should be provided as a list of strings, where each string is a Verilog predicate. In the future, we will exploit invariant synthesis techniques to help synthesize these invariants.

6 Interface Signal Information

The Verilog module comes with a set of I/O signals and the tool needs to know how these signals should be connected. The "interface mapping" field is a dictionary whose keys are the Verilog I/O signals and whose values can be one of the following:

- An ILA input name. This means that the Verilog input signal corresponds to one ILA input. They must have the same encoding and bit-width.
- "**KEEP**" directive. Telling the tool to have a wire of the same name and to connect it as the verification wrapper I/O.
- "**NC**" directive. Indicating that this port does not need to be connected.
- "**SO**" directive. Indicating that this is actually a direct output from a visible state variable (a state variable that is modeled in the ILA).
- "**RESET**" directive. Indicating that this signal is the reset signal (we assume synchronous, active-high reset).
- "**CLOCK**" directive. Indicating that this is the clock signal.
- "**MEM**name.signal" directive. Indicating this signal is the connection to an external/shared memory. The name part should be the ILA state variable name of the memory, and the signal part could be one of the following: "wdata","rdata", "waddr","raddr", "wen", "ren". If the signal does not directly correspond to the write/read data, write/read address, write/read enable signal, it should be specified as "**KEEP**" and in Section 8, you can specify the mapping using the additional assumptions.

7 Uninterpreted Function Mapping

The ILA model may use uninterpreted functions, however, in the verification, the tool must know the correspondence of this uninterpreted function in order to reason about the correctness. In the "functions" field of the `var-map` JSON, a dictionary should be provided if uninterpreted function is used in the ILA model. The keys of the dictionary are the function names, with the values to be a list. Each element of this list is for one application of this function. Per each function application, the tool needs to know the correspondence of the arguments/result in the Verilog and when that happens, this is like a condition/mapping pair, and it is specified as a list. The correspondence of the function result goes first followed by the arguments (in the same order as the arguments in ILA function definitions).

8 Additional Assumptions

This section allows users to add additional assumptions in the verification. They can be

- An assumption about the module I/O.
- A mapping from the Verilog design's memory interface to the provided 6-signal memory interface. The AES case study provides an example of this. The Verilog design uses two signals `stb` and `wr` to indicate memory read and write enable, which are different from the `ren` and `wen` signals. Therefore a mapping is provided as follows:

```
"mapping control" : [
  "( wr & stb) == "__MEM_XRAM_0_wen"  ,
  "(~wr & stb) == "__MEM_XRAM_0_ren"  ]
```