Queries over Variable-Assignment-Model of RTL Systems

Ales Smrcka

November 25, 2015

1 Basic Syntax

The syntax of the query language for traverse through RTL/VAM model is based on LISP. Tokens are either parentheses `(`, `)`, keywords, identifiers, constant numbers, and string literals. The list within parentheses is separated by a white character. Keywords are of two kinds: VAM language keywords and query keywords. See VAM language reference manual for VAM keywords.

2 Query Commands and Keywords

Any LISP-like list represents a query expression. Query expression represents either query command or query object. Query commands are selected by query keywords. Query keyword takes its meaning just on particular places in the query expressions (typically as the first place of a LISP-like list). For instance, when selecting a register identified as `'size'` (which is btw one of the query keywords), the select query looks like: `(select register size)`. This particular query expects `'('` token, then `'select'` keyword followed by another keyword specifying the type of RTL object, i.e. `'register'`, followed by an object identifier, and finally enclosing `')'` token. It does not matter if the register identifier is also a query keyword – query compiler reads it as an identifier.

2.1 Command Commands

`<command_command> ::`

`'print'` Evaluate command parameters (RTL object or subquery) and print them.

```
(print <query1> [<queries> ...])
```

returns printed queries

`'let'` Define a symbolic name for another query or query object. The symbolic name can be seen as variable.

```
(let <var> <query>)
```

returns true if query evaluates to non-empty value.

Note: there are built-in variables representing the set of RTL objects: `signals`, `registers`, `memories`, `memports`, *and* `fnodes`.

`'nop'` No special operation. Just evaluate each of its queries.

```
(nop <query1> <query2> ... <queryN>)
```

returns 1

2.2 Query Objects

Operands of all queries are query objects. A query object is either a symbolic representation of a query or evaluated query in a form of constant. Symbolic representation of a query is either query itself, or a symbolic name (variable) representing it. A constant is string literal, integer, or set or list of constants.

`<query_object> ::`

<var> Identifier specifying bound or free variable. Free variable must be declared by `let` command.

`'set'` Define the set of enumerated items (zero or more items):

`(set <obj1> <obj2> ... <objN>), or`

Define the set of items constraint by <where_query> expression

`(set <query> where <where_query>)`

such that <where_query> must contain a nested query expression which performs as generator for new bound variables referenced by <query>. The simple syntactic sugar is provided for several constaints:

`(set <query> where <q1> <q2> ... <qN>)`

to...

`(set <query> where (and <q1> <q2> ... <qN>))`

Another syntactic sugar deals with `where` keyword being substituted with pipe | symbol. *returns* set object

`'tuple'` Define the ordered list of enumerated items (zero or more items):

`(tuple <obj1> <obj2> ... <objN>), or`

Define the ordered list of items constraint by <where_query> expression:

`(tuple <query> where <where_query>)`

such that <where_query> must contain a nested query expression which performs as generator for new bound variables referenced by <query>. The same syntactic sugar as for `set` query is provided for the `tuple` query.

returns tuple object

<const>, <int>, <bool>, ... Constant is some value of string literal, integer, or bool. For bool constants, each nonempty or nonnull queries are evaluated to true. There is also room for purpose-specific constants (for instance RTL objects). A purpose-specific constant is just a handle of the given object.

2.3 Query Object Operations

<query_object_operations> ::

'add' (add <item_query> <set_query>) => set

'append' (append <item_query> <tuple_query>) => tuple

'not' (not <bool>) => bool

'and' (and <bool1> <bool2> ... <boolN>) => bool

'or' (or <bool1> <bool2> ... <boolN>) => bool

'implies' (implies <boolSRC> <boolDST>) => bool

'at' Accesses index-th item of the list/tuple. Here, index must be integer constant.

(at <tuple> <index>) => list/tuple item

'in' (in <query> <set>) => bool

(in <query> <tuple>) => bool

If <query> is the reference to a bound variable, in performs as generator.

'occurences' (occurences <item_query> <tuple>) => int

returns number of occurences of <item_query> in <tuple>.

'subset' (subset <set or list #1> <set or list #2>) => bool

'intersection' (intersection <set or list #1> ... <set or list #N>) => set

'union' (union <set or list #1> ... <set or list #N>) => set

'bigcap' (bigcap <set or list of sets>) => set

'bigcup' (bigcup <set or list of sets>) => set

'setminus' (setminus <set or list #1> <set or list #2>) => set

'if' (if <bool> <then_query> <else_query)

note, else_query is optional:

(if <bool> <then_query>)

returns expression which is evaluated by <then_query> or <else_query>, or None object if no else branch is provided and <bool> evaluates to false.

'exists' Specify first-order logic formula.

(exists <var> <query1> <query2> ... <queryN>)

returns bool: true if there exists value of variable <var> such that all the queries <queryX> are satisfied.

'forall' Specify first-order logic formula.

(forall <var> <query1> <query2> ... <queryN>)

returns bool: true if for each valid value of variable <var> all the queries <queryX> are satisfied.

'equals', **or**

'=' (equals <expr1> ... <exprN>) => bool

'size' (size <set or tuple>) => int

'empty' (empty <set or tuple>) => bool

'>' (< <int> <int>) => bool

Note that there is no greater-or-equal >= operator. Since the operands are just integers, when expressing a>=b use a>b-1 instead.

'+' (+ <int> <int>) => int

'-' (- <int> <int>) => int

2.4 RTL Objects

An RTL object represents just a special constants of queries. The following keywords identify the type of a given object.

<type_of_rtl_object> ::

'signal'

'register'

'memory'

'memport'

'fnode'

'expr'

There are classes of RTL objects:

<rtl_object_classes> ::

'<rtl object>' Represents RTL object of any class.

'<storage>' Represents RTL storage, like register, memory, or memory port.

'<rtl node>' Represents storage or functional node.

2.5 RTL Object Operations

<rtl_object_operations> ::

'select' *Deprecated, to be removed in the future.* Find an RTL object wrt. its identifier.

(select <type_of_rtl_object> <id>)

returns handle of the given RTL object.

'select-signal'

'select-register'

`'select-memory'`

`'select-memport'`

`'select-fnode'` Find an RTL object wrt. its identifier.

```
(select-signal <string>)

(select-register <string>)

(select-memory <string>)

(select-memport <string>)

(select-fnode <string>)
```

returns the given RTL object.

`'is-instance' (is-instance <rtl object> <type_of_rtl_object>) => bool`

`'expr-type' (expr-type <rtl object of expression type> <vaml operator>) => bool`

`'rtlid' (rtlid <rtl object>)`

returns string, identifier of the given object

`'inputs' (inputs <rtlnode>) => set`

returns set of signals being inputs of the node.

`'inputs' (inputs <signal>) => set`

returns set of RTL expressions (zero or one assignment) and storages to which the signal is connected as their input.

`'inputs' (inputs <rtl object of expression type>) => set`

returns set of signals influencing the value of the expression.

`'outputs' (outputs <rtlnode>) => set`

returns set of signals being outputs of the node.

`'outputs' (outputs <signal>) => set of expressions`

returns set of RTL expressions which reference the given signal.

`'rtloutputs' (rtloutputs <signal>) => set of expressions or storages`

returns set of RTL expressions which reference the given signal and storages being directly influenced by the signal.

`'references' (references <set of expressions>) => set of signals`

returns set of signals referenced by expressions.

`'path' (path <set of input signals> <set of output signals>)`

returns tuple of expressions

`'direct-path' (direct-path <set of input signals> <set of output signals>)`

returns tuple of expressions

`'signal-path' (signalpath <set of input signals> <set of output signals>)`

returns tuple of signals

`'coi' (coi <no of cycles> <set of signals>) => set of signals`

`'coib' (coib <no of cycles> <set of signals>) => set of signals`

`'targets' (targets <signal>) => set of storages`

`'sources' (sources <signal>) => set of storages`

`'data-in' (data-in <storage>) => signal`

`'data-out' (data-out <storage>) => set of signals`

`'address' (address <storage>) => signal`

`'enable' (enable <storage>) => signal`

`'bitwidth' (bitwidth <rtl object>) => int`

returns bitwidth of the given RTL object. In a case of register or port, returns bitwidth of its data signal.

`'initvalue' (initvalue <register>) => int`

returns initial value of the register (initial value is the value when resetting the register)

`'read-port' (read-port <memory port>) => bool`

`'write-port' (write-port <memory port>) => bool`

`'getattr' (getattr <rtl object> <attribute name>) => string`

returns attribute (meta) value of the given RTL object and attribute name (string)

`'setattr'` Sets the value of the attribute for the given RTL object.

`(setattr <rtl object> <attribute name> <attribute value>) => int`

returns always 1

`'unsetattr'` Resets (delete the meta information) attribute of the given RTL object.

`(unsetattr <rtl object> <attribute name>) => in`

returns always 1

`'meta-clear'` Clears all metadata of the given RTL object.

`(meta-clear <rtl object>)`

returns always 1

2.6 System Manipulations

System manipulations are purpose-specific query commands. These are useful for scripting.

`<system_manipulation> ::`

`'load'` Loads RTL system from the file. All system variables are initialized wrt. the system. All the variables dependent on these sets are invalidated.

```
(load "filename")
```

returns 1

`'save'` Saves (export) RTL system to the file.

```
(save "filename")
```

returns 1

`'simplify'` Simplifies the system to reduce the number of signals, registers, functional nodes, and uniforms the expressions. All the variables are cleared and system variables (like signals, registers, memports, memories, and fnodes) are initialized wrt. a new, simplified, version of the system.

```
(simplify)
```

returns 1

`'unifyselectors'` Unifies all selectors (ITE + multi-conditional expressions + switch expressions) into simple switch expressions over *selectors* (a variable which value represents a case). Each switch expression is encapsulated to a single functional node. A RHS of each switch-case is either a constant or a variable reference (i.e., there are no nested expressions).

```
(unifyselectors)
```

returns 1

`'load-vars'` Loads all the system objects (signals and storages) and make them variables. The new variables will be identified by their identifiers, each with the prefix "s_" for signals, "r_" for registers, "m_" for memories, and "p_" for memory ports.

```
(load-vars)
```

returns int, number of new variables

3 Variables Declaration and Validity

Each variable is declared either by `'let'` or by `'in'` command (note the `'in'` query sometimes performs as value generator). The `'let'` command has the priority over `'in'` for declaration, thus when used like this:

```
(let x 42) (in x signals)
```

the command `'in'` performs as query object operation ("Is `x` in the set `signals`?"). But when used like this:

```
(exists x (let y (outputs x)) (in x signals))
```

the variable is declared by the command `'in'` and is bound with existential operation `'exists'`.

Since the variable value depends on values of other variables (wrt. declaration of the variable), the variable value is valid as long as all values the variable depends on are valid. If a bound-variable does not depend on anything, its value is valid only within the query responsible for declaration of bound-variable. If a free-variable does not depend on anything, its value is valid forever. For instance:

```
(let y (size (signals)))
(exists x (in x signals) (let z (outputs x))
```

Value of variable `x` is valid just within `exists` query. Value of variable `z` is valid as long as value of `x` is valid. Value of `y` is valid as long as `signals` is valid, and since `signals` are valid untill the RTL system changes, `y` is valid till then.

4 Examples

Example 1: Get all registers which do not have enable signal:

```
(set r where
    (in r registers)
    (empty (enable r))
)
```

Example 2: Get all pairs of different registers:

```
(set (tuple ra rb) where
    (in ra registers)
    (in rb registers)
    (not (equals ra rb))
)
```

Example 3: Get all signals connected to address port of a memory.

```
(set addr where
    (exists mem (in mem memories) (in addr (address mem)))
)
```

Example 4: Find a program counter which is connected to address signal of a memory. Find an instruction decoder (instruction vector) register which is filled by the data-out of above mentioned memory and which is connected to at least three bitvector slicing expressions (one for opcode, two for operands).

```
(set (tuple pc id) where
    (in pc registers)
    (in id registers)
    (exists mem
        (in mem memports)
        (read-port mem)
        (path (data-out pc) (address mem))
        (path (data-out mem) (data-in id))
    )
    (let usages (bigcup (set (outputs osig) where
        (in osig (data-out id)))))
    (>
```

```
            (size (set e where
                (in e usages)
                (expr-type e [..])))
            2
        )
    )
```

Example 5: Get all signal paths between each pair of registers.

```
(set (signal-path sa sb) where
    (exists ra (and (in ra registers) (in sa (outputs ra))))
    (exists rb (and (in rb registers) (in sb (inputs rb))))
)
```

Example 6: Get the set of signals which influence maximum number of other signals.

```
(set a where (forall b
        (in a signals)
        (in b signals)
        (or
            (equals a b)
            (> (size (coi 1 a)) (- (size (coi 1 b)) 1))
        )
))
```

Example 7: Set the program counter as active only for the 1st stage.

```
(meta-clear (select-register "pc"))
(setattr (select-register "pc") "stage-fe" 1)
```

Example 8: Selects all signals active before EX (3rd) stage (requires appropriate settings of the attributes).

```
(set s | (in s signals)
    (or
        (getattr s "stage-fe")
        (getattr s "stage-id")
        (getattr s "stage-ex")
    )
)
```

5 Undocumented Commands

Here are some commands which are intended to be used for debugging purposes. Their behavior may lead to unexpected results of normal queries.

`'debug-vars'` Prints the information about all variables.

```
(debug-vars) => string
```

`'undef'` Undefines the variable.

```
(undef <varid>) => bool
```

returns true if the variable has been undefined. false if the variable has some dependencies.

`'debug-setup'` Setups variables and variable groups wrt. query. Does not evaluate the query. Starts debugging mode in which bound variables are not automatically erased.

```
(debug-setup <query>) => none
```

`'debug-varsdep'` Prints variable dependency graph. The `dot` graph is printed to a given file. Solid-framed nodes are free variables, dashed nodes are temporary variables, and bold nodes are bound variables. An edge from node A to B represents that value of variable A depends on the value of variable B. A dotted edge represents dependency on bound variables.

```
(debug-varsdep "deps.dot") => string "written to deps.dot"
```

`'debug-var'` Prints the information of the given variable.

```
(debug-var <id>) => string
```

`'debug-vgroups'` Prints the information about variable groups.

```
(debug-vgrous) => string
```

`'debug-next'` Computes the next value of the bound-variable.

```
(debug-next <varid>) => variable value
```

`'debug-stop'` Stops debugging mode and clear all structures about temporary variables.

```
(debug-stop) => none
```

`'system'` Runs shell command.

```
(system "command to be executed by system shell") => string of stdout
```