

Vapor User Guide (v0.95)

Zaher Andraus

Department of Electrical Engineering and Computer Science

University of Michigan, Ann Arbor

October 2004

1.0 INTRODUCTION

Vapor stands for Verilog Abstraction for Processor Verification. In Vapor, RTL Verilog can be “vaporized” (abstracted) to term-level model for verification purpose. The resulting model can be analyzed with validation engines like UCLID [2] and SVC [7].

Vapor is part of the Reveal package, which is based on the abstraction/counterexample-guided refinement flow, and is releases separately.

Vapor main algorithm is datapath abstraction based on Verilog syntax. It translates behavioral and synthesizable Verilog RTL to UCLID, in which datapath is abstracted away and modeled efficiently via quantifiers-free first order logic.

Vapor main components are:

1. Verilog-to-UCLID engine which generates UCLID code from the Verilog and brief specifications. It mainly abstracts the datapath by black boxing Verilog operators and representing bit-vectors as integers from unbounded domain.
2. A Verilog syntactical flattener which is responsible for generating hierarchies-free Verilog model without performing any synthesis operation.
3. A Verilog-to-Verilog preprocessing utility which allows to specify some bit-vectors to remain unabstracted.

Please refer to [5] for elaborate explanation on UCLID and its verification paradigm.

Throughout this document we will use UF and UP to stand for Uninterpreted Function and Predicate, respectively.

2.0 VAPOR’S VERILOG TO UCLID ENGINE

Vapor performs Verilog-to-UCLID abstraction based on syntactical considerations. Verilog bit-vectors are abstracted into UCLID terms, while 1-bit signals are translated into truth variables. Verilog operators that are applied on bit-vectors map into uninterpreted functions and predicates, while Boolean operators map to Boolean UCLID operators.

2.1 Input

2.1.1 Verilog

Vapor accepts Synthesizable Verilog, which complies with the input requirements of Icarus Verilog front-end [3,4]. The Verilog code has also to follow these rules:

1. All signals have to be initialized.
2. The top module (and only this module) is not allowed to have any ports. Consequently, output ports has to be converted to internal wires or registers, and inputs has to be modeled as wires or registers and their behavior has to be explicitly defined like any other Verilog signals. To model non-deterministic inputs, use non-deterministic initialization and next-state functions (see *extended syntax* below). Future versions of Vapor may waive the previous requirement from inputs and to automatically assume that inputs are registers that are non-deterministically initialized and next-state'd.
3. The Verilog code should not include any non-supported construct according to Section 2.11.
4. The code should not exercise the semantics of *blocking assignment*. The occurrence of it is indeed allowed, however Vapor treats as if it is *non-blocking*. However, Vapor does allow “default value” statement as a procedural assignment (see *extended syntax* below).
5. The design should be implicitly synchronized using a single clock (edge).
6. All memory (2-D) arrays (as well as any bit-/part-selection that is not constant-indexed) have to be removed and modeled via memory elements that are defined in the *directives file* (see next section)

Verilog Extended Syntax

Our light extension to Verilog syntax includes the following:

1. Signals can be non-deterministically (symbolically) initialized and next-state'd. The token *TD_B* (*ND_T*) is preserved for symbolic fresh Boolean (Term) value. For example, to define the behavior of a Verilog bit-vector as non-deterministic, do the following:

```
reg [7:0] vect;  
initial vect := ND_T;  
always @(posedge clk) vect = ND_T;
```

2. Since blocking assignments are widely used for specifying default values in procedural blocks, we have extended the syntax as follows: Default value statements has to be written as *DEFV signal = value*, and each signal has to have only one such statement per procedural block.

2.1.2 Directives

Vapor also requires a *directives file* for specifying the configuration of the verification performed on the given design. The syntax of the directives file is as follows:

```

.clock <clock_signal>
.clock_edge up/dn
.mem <owner module> <instance name> {{<data signal> <address signal> [write-enable signal]
    in/out delayed/comb}*}*
.ignore <module>
.top <top module>
.eqc <module1> <module2>
.corr_ch <spec depth> <spec stall> <pipeline depth> <pipeline stall>
    {<memory module1> <memory module2> pc/wr/rd}*
.axiom <module> <axiom signal> up/dn
.reset <reset signal>
.prop <property signal>
.bmc <number of cycles>

```

Figure 1. Directives File

There is no meaning to the order of the directives, however some directives are obligatory, and some are mutually exclusive with others. *clock* and *clock_edge* are the only mandatory directives and they specify the implicit clock used in the design (explicit clock is not supported). This is the only signal that Verilog should contain edged-events for. Using events that “guard” a different edge (different signal, or different edge for the clock) is an error.

The directives *eqc*, *corr_ch*, and *bmc* are used to specify the *verification method*, which is explained in Section 2.8. The *top*, *axiom*, *reset*, and *prop* directives are used to specify respectively: the *top module*, some axiom signal (as many as you want), the *reset signal*, and the *property signal*. The *property signal* should be a 1-bit Verilog signal that is declared in the *top module*, and this signal is validated to be 1 in UCLID. Vapor allows axioms to hold for the UCLID verification, and such are defined via *axiom*. The *ignore* directive is used to specify modules that should be ignored while processing the Verilog for abstraction. Section 2.9 will elaborate on modeling of memories via the *.mem* directive.

2.2 Abstraction of Verilog Signals

Since Verilog allows extraction and concatenating of bit-vectors, modeling bit-vectors as UCLID terms necessitates maintaining consistency between terms that correspond to different parts of the same bit-vector, for sound abstraction. These consistency issues are elaborated on in [1], and an example is given in Section 2.13.1. Verilog 2-D arrays are not allowed in the Verilog code, and should be modeled as memories (i.e. specified in the *directives file*).

2.3 Abstraction of Verilog Constants

As explained in [1], Vapor abstracts constants that are greater than a constants threshold, while models smaller constants via the interpreted *succ* function applied on the zero term *UCLID_CONST_0*. It is possible to manually specify the *constant threshold*, or invoke a preprocessing scheme that will compute the highest threshold that will not compromise the encoding. Abstracted constant may lead to false negatives (dealt with in Reveal), while big unabstracted constants may lead to significant increase in the size of the encoded CNF model (from UCLID) and loss of the advantage gained from datapath abstraction.

2.4 Abstraction of Verilog Operators

Verilog operators map to UCLID Boolean operators, UFs and UPs. Addition and subtraction of “small” constants maps to *succ* and *pred* accordingly. After introducing a general naming convention for UCLID variables (and UFs/UPs) in Section 2.6, we will bring (Section 2.7) an elaborate table for the Verilog-to-UCLID operators mapping.

2.5 Type Casting

In some cases Vapor needs to cast a term variable into a truth, or vice versa. For example, when concatenation of *bits* is involved, and particularly given that a concatenation UF *has* to accept terms only (UCLID syntax rules), a truth-to-term conversion is needed. This is done by defining a term replacement variable that is equal to *UCLID_CONST_1* if and only if the truth variable is value is 1. On the other hand, term-to-truth conversion is done using a global casting *Lambda Expression*.

2.6 Naming Convention

Next, we will introduce Vapor’s naming convention for the term variables, UFs and UPs, which will facilitate the abstraction of the example given in the following section.

1. *<name>_field_n_m* stands for a UCLID variable that models a sub-field of the bit-vectors *name*, i.e. *name[n:m]*.
2. *<name>_sub_field_n_m* stands for a UCLID variable that models a partition [1] sub-field of bit-vector *name*, e.g. the induced sub-fields that is created by *name[5:3]* and *name[4:0]* are represented as *name_sub_field_5_5*, *name_sub_field_4_3*, and *name_sub_field_2_0*.
3. *Sel_k_l* stands for an extraction UF/UP starting position *k* and extracting *l* bits. *Sel_4_2(IR)* for example stands for the extraction *IR[5:4]*.
4. *Concat_n1...nk* stands for concatenating *k* terms of sizes *n1 ... nk*.
5. *UCLID_CONST_k* stands for a constant that has the semantics of the integer *k*, i.e. *k* integer units from the globally defined zero constant, *UCLID_CONST_0*.
6. *UCL_<op>* stands for a UF/UP that models a Verilog operator. For example, *UCL_BITW_NOT* stands for bit-wise inversion UF.
7. *NDTerm* and *NDTruth* are used for generation of non-deterministic terms and Booleans. Vapor represents random (non-deterministic) terms with *NDTerm_init*, and generates new ones by *NDTerm_succ(<previous non-deterministic value>)*. For Booleans, Vapor uses *NDTruth_init* and uses *NDTerm_pred* to generate the truth out of the non-deterministic term.
8. *concat_rep* to represent a concatenation expression that was replaced in the Verilog code.
9. *UCL_TMP* to represent temporary place holders for capturing current/next state expressions of certain UCLID variables.

2.7 Mapping List

The next list includes a mapping from Verilog bit-vector operators to UCLID UFs and UPs.

TABLE 1. Map of Verilog Operators to UCLID UFs/UPS

Verilog Operator	UCLID	Type	Verilog Operator	UCLID	Type
*	UCL_MULT	UF	== , ===	=	Equality
/	UCL_DIV	UF	Unary &	UCL_BITW_U_AND	UP
%	UCL_MOD	UF	Unary ~&	UCL_BITW_U_NAND	UP
<<	UCL_SL	UF	Unary	UCL_BITW_U_OR	UP
>>	UCL_SR	UF	Unary ~	UCL_BITW_U_NOR	UP
+	UCL_ADD	UF	Unary ^	UCL_BITW_U_XOR	UP
-	UCL_SUB	UF	Unary ~^	UCL_BITW_U_XNOR	UP
~	UCL_BITW_NOT	UF	<	UCL_LESS	UP
&	UCL_BITW_AND	UF	≤	UCL_LESS_EQ	UP
	UCL_BITW_OR	UF	>	UCL_GTR	UP
^	UCL_BITW_XOR	UF	≥	UCL_GTR_EQ	UP
~^	UCL_BITW_XNOR	UF			

2.8 Verification Methods

In UCLID, one can freely customize the verification method (see the *CONTROL* module description in [5]). In the current version of Vapor, however, this is not the case. For the purpose of automation, Vapor supports a limited number of verification methods that are well defined, although parametrized. This section delves into each one of them.

2.8.1 Bounded Model Checking (BMC)

An *initial* process is mandatory for this method (at least *one* such process), and all Verilog signals has to be initialized. The *directives file* has to supply (via the *.bmc* directive) the number of cycles the model should be unfolded and checked. The property to be checked at the k-reachable state (for k-BMC) can be specified via the *.prop* directive.

2.8.2 Invariant Checking (IC)

A rather special case of *BMC*, where the number of cycles to unfold is 1. However, in most *IC* verifications one would like to assume correctness in the initial state. This is feasible via the *.axiom* directives.

2.8.3 Correspondence Checking (CC)

Correspondence Checking [6] is used for verifying that a pipelined implementation complies with a non-pipelined version, often called the *specification* module. To specify *CC* for Vapor, one has to use the *.corr_ch* directive and specify:

1. The spec depth,
2. The spec stall signal,
3. The pipeline depth,
4. The pipeline stall signal
5. And the corresponding modules.

The need to specify depth for the spec implies that Vapor can prove equivalence of 2 pipelined machines.

The *.eqc* specifies equivalence of the two modules (implementation and specification), and the corresponding sub-modules in these 2 modules are enumerated in the *.corr_ch* directive. Each corresponding pair of sub-module is either:

1. A writable memory module (*wr*), i.e. a module that the pipeline and specification write to (and optionally read from), and has to be checked for consistency among them,
2. A read-only memory module (*rd*), i.e. a module that the pipeline/specification read from, but do not write to; such module is not checked for correspondence (da! because it is fixed), but both modules (the one in the specification and the one in the implementation) are initialized with the same symbolic value,
3. A PC register (*pc*), that both the specification and implementation have. For microprocessors, this register represents the instruction load address, and has special treatment in *CC* [6].

Section 2.13.3 will bring a compact representative *CC* example.

2.9 Memories

Designers often define memory signals (2-D arrays). Any Verilog array should be treated as memory, even arrays of 1-bit wide. Memories can be potentially accessed symbolically, and they are always modeled in UCLID via UFs and UPs. A memory module has:

1. An address signal,
2. A data signal,
3. Optionally, a write-enable signal,
4. A direction,
5. and a type.

These components are defined in the relevant directive in the *directives file*. The direction specifies whether it is a port for reading (out) or writing (in). The type of port specifies whether (relevant only for *in* ports, *out* ports are always *comb*) the writing is 1-cycle delayed (delayed) or happens in the same cycle

(comb) since the write-endable allows that. *comb* memory is useful for memories that latch on the negative edge to allow forwarding.

In the current version of Vapor, both data and address signals of the memory has to be bit-vectors (only UFs).

2.10 Customized Abstraction

Section 3.3 will talk about selective abstraction of bit-vectors. Vapor also allows controlling the abstraction of certain modules. This, however, was omitted from the current version due to the flattener that was introduced into the flow (this also means that the *.ignore* directive is not consistent with the functionality of the flattener). Reenabling such capability is straightforward, and we plan to include it in the new release. It will allow modeling of combinational and sequential black boxes as uninterpreted functions.

2.11 Non-Supported constructs

We currently do not support the following:

1. Passing arguments by name matching, when instantiating modules. Also, we do not support passing expressions instead of signal names for instantiation.
2. Concatenation expressions with non-constant repeat argument, shifting with non-const argument, or selection with non-const indices (the last case can be modeled via memory...)
3. Tasks and functions.
4. *release*, *trigger*, *while repeat*, *noop*, *for*, *forever*, *force*, *deassign*, *disable*, *casex* and *casez*.
5. A concatenation expression as a LHS of an assignment.

Some of these constructs are not supported because of implementation burden. If you can justify its need, we can implement that in future release.

2.12 Configuring Vapor

One of the configuration files of Vapor is called the *config* file (look for *config_standard* as well). This file specifies:

1. The constants threshold
2. Whether to preprocess the Verilog or not (*no* has to be specified always!)
3. Whether to flatten the Verilog or not (*no* has to be specified always, since flattening is done by an external scrip!)
4. The file name to which Vapor will dump the sizes of certain signals
5. Whether to include all extraction/concatenation axioms. Use this option to control the presence or absence of such axioms, in case you have an insight on the Verilog coding style. For example, if the axiom that guarantees that concatenating sub-fields will yield the original bit-vector is not needed, use the *axioms no* option. The abstraction will still be sound, although false negatives may show up in case such axioms were actually needed.

6. The *compile_only* option specifies the “translation mode” of Vapor. A *yes* option directs Vapor to invoke the Icarus Verilog compiler without generating any UCLID code. A *no* option invokes the UCLID generation. A *divide* option with the divisions file name will invoke the Verilog preprocessor that dumps a divided (sliced) bit-vectors (see *Section 3.3*). Other options are possible and no documentation is currently available for those.

2.13 Abstraction Examples

2.13.1 Bit-vectors Consistency

We bring a small example that will raise most of the issues we think are important for Vapor users, and is given in the installation package (*example9.v*). Consider the following code given in Figure 3.

```

1  reg [16:0] word;// 17-bit register
2  wire [7:0] w_low;// 8-bit bus
3  wire [7:0] w_high;// 8-bit bus
4  wire [16:0] out;// 17-bit bus
5  wire parity;// single-bit wire
6  wire clk;    // clock
7  reg mode;    // single flip-flop
8  always @(posedge clk)
9    if (mode == 1'b1)
10     word[10:3] <= 8'b11001110;
11  else
12     word <= {parity,{w_high,~w_low}};
13  assign out = word;

```

Figure 2. Extraction Example - Verilog

For this code, Vapor will create term variables for *word*, *w_low*, *w_high* and *out*, and truth variables for *parity* and *mode*. As described in [1], Vapor will “standerize” the assignment in *line 12* which has concatenation, and convert it into assignments to the bit-fields of the signal *word*. Thus, the first pass will yield the following code:

```

1  reg [16:0] word;// 17-bit register
2  wire [7:0] w_low;// 8-bit bus
3  wire [7:0] w_high;// 8-bit bus
4  wire [16:0] out;// 17-bit bus
5  wire parity;// single-bit wire
6  wire clk;    // clock
7  reg mode;    // single flip-flop
8  wire [16:0] concat_rep;
9  assign concat_rep = {parity,{w_high,~w_low}};
10 // assign concat_rep[7:0]    = ~w_low;
11 // assign concat_rep[15:8]  = w_high;
12 // assign concat_rep[16:16] = parity;
13 always @(posedge clk)
14 if (mode == 1'b1)
15     word[10:3] <= 8'b11001110;
16 else
17     word <= concat_rep;
18 assign out = word;

```

Figure 3. Extraction Example - Preprocessed Verilog

The assignment in *line 9* is simply a wire that is equal to the RHS of the original assignment in *line 17*. The assignments in *line 10* through *line 12* are the interpretation that Vapor gives to any assignment in general, which has RHS to be a concatenation expression.

The final resulting UCLID code can be obtained by running Vapor on *example9*. However, we will bring here the main segment of that code, that reflect the assignments to the bit-vector *word*:

```

1  DEFINE
2  UCL_TMP_0                := word_sub_field_2_0;
3  UCL_TMP_1                := UCLID_CONST_BIG1;
4  UCL_TMP_2                := word_sub_field_16_11;
5  concat_rep_field_7_0    := UCL_BITW_NOT(w_low_field_7_0);
6  concat_rep_sub_field_7_0 := UCL_BITW_NOT(w_low_field_7_0);
7  concat_rep_field_15_8   := w_high_field_7_0;
8  concat_rep_sub_field_15_8 := w_high_field_7_0;
9  concat_rep_field_16_16  := parity_field_0_0;
10 concat_rep_sub_field_16_16 := parity_field_0_0;
11 UCL_BOOL_representative := case
12   concat_rep_sub_field_16_16 : UCLID_CONST_1;
13   default: UCLID_CONST_0;
14 esac;
15 concat_rep_field_16_0    := Concat_8_8_1(concat_rep_sub_field_7_0,
16                                           concat_rep_sub_field_15_8,
17                                           UCL_BOOL_representative);
18 UCL_TMP_3                := Sel_3_8(concat_rep_field_16_0);
19 ASSIGN
20 next[word_field_10_3]    := case
21   mode_field_0_0 : UCLID_CONST_BIG1;
22   ~mode_field_0_0 : UCL_TMP_3;
23   default: word_field_10_3;
24 esac;
25 next[word_field_16_0]    := case
26   mode_field_0_0 : Concat_3_8_6(UCL_TMP_0,UCL_TMP_1,UCL_TMP2);
27   ~mode_field_0_0 : Concat_rep_field_16_0;
28   default: word_field_16_0;
29 esac;
30 next[word_sub_field_10_3] := case
31   mode_field_0_0 : UCLID_CONST_BIG1;
32   ~mode_field_0_0 : Sel_3_8(concat_rep_field_16_0);
33   default: word_sub_field_10_3;
34 esac;
35 next[word_sub_field_16_11] := case
36   ~mode_field_0_0 : Sel_11_16(concat_rep_field_16_0);
37   default : word_sub_field_16_11;
38 esac;
39 next[word_sub_field_2_0]  := case
40   ~mode_field_0_0 : Sel_0_3(concat_rep_field_16_0);
41   default: word_sub_field_2_0;
42 esac;

```

Figure 4. Extraction Example - UCLID

Vapor uses *UCL_TMP* variables to assign names (similar to “pointers”) to certain needed expressions. Due to the assignment to *concat_rep* (*line 9* in *Figure 3*), it was divided into 3 partition sub-fields. The bit-vector *word*, however, was divided into the partition {[16:11],[10:3],[2:0]}. *UCLID_CONST_BIG1* is a con-

stant Term that represents the Verilog constant, and it was not defined as some successor over *UCLID_CONST_0*, since it was bigger than the default constants threshold. The example above shows modeling of both sequential and combinational updates to bit-vectors with selection. Enjoy!

2.13.2 Memory

Memory modeling is very important since, firstly, it encapsulate a big “volume” of the datapath, thus reduces the state space tremendously. Secondly, memory modeling has to be done precisely and maintain equivalent semantics to their Verilog description.

For example, consider the following memories directive:

```
.mem Mod MyMem {{Dout Addr out comb} {Din Addr RE in delayed}}
```

This directive will create the following UCLID code:

```

1  VAR
2  UclMem_MyMem0_aux : FUNC[1];
3  UclMem_MyMem0    : FUNC[1];
4  CONST
5  UclLambdaSelTerm : TERM;
6  UclMem_MyMem0_aux_INIT : FUNC[1];
7  DEFINE
8  UclMem_MyMem0 := UclMem_MyMem0_aux;
9  Dout_field_7_0 := UclMem_MyMem0(Addr_field_7_0);
10 ASSIGN
11 init[UclMem_MyMem0_aux] := UclMem_MyMem0_aux_INIT;
12 next[UclMem_MyMem0_aux] := Lambda(UclLambdaSelTerm).case
13   Addr_field_7_0 = UclLambdaSelTerm & RE_field_0_0 : Din_field_7_0;
14   default: UclMem_MyMem0_aux(UclLambdaSelTerm);
15 esac;

```

Figure 5. UCLID code with memory

In the current implementation, the directive should reference the exact name of the UCLID variables. For example, the directive should include *Dout_field_7_0* instead of *Dout*. This inconvenient restriction will be removed in the coming release. Additionally, since Vapor declares UCLID variables only when the corresponding bit-vectors are being used, make sure you use the memory output, i.e. pass its value to another bit-vector wire (*assign Dout_final = Dout;*).

Vapor uses the auxiliary memory component in order to allow modeling of same-cycle writing to memory (for example, memories that latch in the negative edge of the clock). Once you replace the *delayed* with *comb*, the code will be different in one assignment only:

```

1  next[UclMem_MyMem0] := Lambda(UclLambdaSelTerm).case
2   Addr_field_7_0 = UclLambdaSelTerm & RE_field_0_0 : Din_field_7_0;
3   default: UclMem_MyMem0_aux(UclLambdaSelTerm);
4  esac;

```

Figure 6. UCLID code with memory that latches updates without a cycle delay

This example is brought in *example10.v* in the installation package.

2.13.3 Correspondence Checking

example11 in the *installayion* package includes a very simple “pipelined XOR”. It includes a specification and the pipelined implementation. We will not bring the Verilog nor the UCLID code here, but we will explain some issues...

Vapor will inject the variables *impl_flush*, *spec_flush*, *proj*, and *isa* to the implementation and specification modules, in order to control the execution of each module. As explained in [6], the implementation is initialized with arbitrary symbolic values, and executed or flushed accordingly. In general, with the presence of complex forwarding logic, some invariants are needed to insure consistent state of the pipeline at cycle 0. The current release of Vapor does not allow adding these axioms externally, and the user is asked to add them to the UCLID code. However, the task can be easily simplified by specifying place holders in Verilog (regular bit-vector registers or wires), which will capture the state of some other Verilog signals that are involved in the required invariants. We are planning to extend Verilog syntax to allow reasoning about the signals in various cycles of the symbolic execution.

3.0 PREPROCESSING IN VAPOR

3.1 Modules Flattening

Vapor allows multiple modules in the design, and can flatten them to retrieve a single flat module. However, unlike Verilog flattening that most tools perform, our flattening does not perform any “reshaping” to the Verilog code. It is a mere syntactical substitution of instantiated modules to create a single flat module, without any synthesis-related operation.

3.2 Concatenation Standardization

As explained in [1], Vapor standardize all concatenation expressions to create only explicit sub-field updates. This is done by a preprocessing step that replaces all concatenation expressions that appear through the code with place holders, and define those place holders via continuous assignments, which will be internally converted into explicit update to sub-fields of the place holder.

3.3 Bit-vectors Division

Vapor uses Verilog rewriting in order to allow modeling bit-vectors bit-arrays. This is done by dumping a Verilog code that will create bit-arrays for every signals that is specified in the *devisions list*. This includes substituting bit-array instead of bit-vectors, and synthesizing Verilog operators that are being applied on such arrays. All Verilog operators can operate on bit-arrays, except for multiplication. The division list is a list of pairs, separated by white spaces. Each pair specifies the relevant module and signal name to be divided. Such list can be written in any file and the file name has to be specified in the *config* file (see Section 2.12).

REFERENCES

- [1] Zaher S. Andraus and Karem A. Sakallah, “Automatic Abstraction and Verification of Verilog Models”, DAC 2004.
- [2] Shuvendu K. Lahiri, Sanjit A. Seshia and Randal E. Bryant, “Modeling and Verification of Out-of-Order Microrprocessors in UCLID”, FMCAD 2002.
- [3] “IEEE Std 1364-1995, IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language” IEEE Incs.
- [4] www.icarus.com/eda/verilog/
- [5] Sanjit A. Seshia, Shuvendu K. Lahiri, Randal E. Bryant, “A User’s Guide to UCLID version 1.0”
- [6] J. R. Burch and D. L. Dill, “Automatic Verification of Pipelined Microprocessor Control”, CAV 1994.
- [7] C. Barrett, D. Dill, and J. Levitt, “Validity checking for combination of theories with equality,” POPL 2002, January 2002, pages 1-3.