A Strategy Language for Testing Register Transfer Level Logic

Michael Katelman and José Meseguer
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801 USA
{katelman,meseguer}@uiuc.edu

***Abstract*—The development of modern ICs requires a huge investment in RTL verification. This is a reflection of brisk release schedules and the complexity of contemporary chip designs. A major bottleneck to reaching verification closure in such designs is the disproportionate effort expended in crafting directed tests; which is necessary to reach those behaviors that other, more automated testing methods fail to cover. This paper defines a novel language that can be used to generate targeted stimuli for RTL logic and which mitigates the complexities of writing directed tests. The main idea is to treat directed testing as a meta-reasoning problem about simulation. Our language is both formalized and prototyped as a proof-search strategy language in rewriting logic. We illustrate its novel features and practical use with several examples.**

I. Introduction

Methodologies for register transfer level (RTL) logic verification typically involve a mixture of different techniques and tools, including simulation, model checking, and equivalence checking, among others. Employing a variety of tools from this ecosystem produces the best results, but simulation-based techniques still dominate the overall process. *Constrained-random* simulation involves generating concrete stimulus by randomly sampling from all possible inputs, modulo a set of user-given constraints. As a result of research work, *e.g.*, [1]–[4], and through the development of hardware verification languages (HVLs) supporting it, the constrained-random paradigm is used extensively and is able to efficiently cover much of a verification plan. However, the ability for constrained-randoms to find new behaviors usually levels off after some time, and what is left is a gap where, say, 10% of the simulation test plan must be closed through other means. Lacking effective means for doing *directed testing* therefore creates a *verification bottleneck* where much more than 10% of the verification effort is spent closing the final 10% of testing obligations.

Alternatives to directed and constrained-random testing, *e.g.*, [5], do help alleviate the problem; but as long as automated methods fail to close testing entirely, there will be an important place for directed testing. In order to avoid being a bottleneck, engineers need tools for crafting directed tests that rely on a balance of *high-level knowledge* provided by the engineer and *automation* provided by the computer. Indeed, what is frustrating for an engineer is that while he/she may have a clear notion of *how* a test should be constructed, *actually writing* a directed test by hand is a tedious and error prone proposition. Consider a microprocessor such as the standard 5-stage "DLX" pipeline from [6] (see Fig. 1), and suppose that we desire a test where a load to the data cache is to a line that has recently been pushed out of the cache and up the memory hierarchy, but has not made it to its destination. The engineer will realize immediately that a test has to be created with some locality and conflict in the memory reference stream, but getting the latencies and addresses right will be extremely difficult. Currently, engineers either write the test entirely by hand, or they modify their random constraints to achieve a more directed test.

We show in this paper that by making *symbolic simulation a first-class object* that can be manipulated, stored, and used programmatically, directed testing is made easier and more effective. We formally define a language having this feature and demonstrate through examples how it can alleviate the most tedious aspects of crafting directed tests. In the example above, the engineer is hindered in two ways, one being the difficulty in knowing *when* the cache eviction will occur, and the second being the difficulty in knowing to *which address*. Our language allows the user to look at the *future state* of the simulation to resolve the first problem, and has a sophisticated interface for generating and solving symbolic execution instances, thereby aiding in the second problem. In Section III we provide several examples of directed tests with simple strategies that use these features to good effect.

Formally, we treat the problem of crafting a directed test as an exercise in *proof strategy creation*. A strategy in the context of this paper is taken to mean a way of organizing proof search; exactly in the same sense that ML [7] was constructed for orchestrating proof search in LCF. Before the advent of ML, proving a theorem $\Gamma \vdash \varphi$ meant laboriously applying the inference rules of LCF by hand to justify φ from the axioms Γ. ML, or *meta-language*, made proofs *first-class* objects that could be manipulated, stored, and used within a general program searching for a particular proof of interest; and made constructing proofs in LCF much easier. In the context of *formal verification* of hardware, languages such as ACL2 [8] and *reFLect* [9] also take advantage of this idea in order to reason about how a hardware design behaves. In this paper we show that, like formal verification, directed testing is made easier by adding a meta-reasoning layer where symbolic simulation becomes a first-class object. The judgments in our

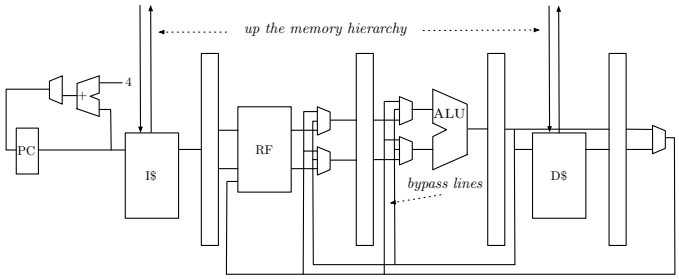

Fig. 1. A Sketch of the DLX Pipeline.

strategy language are elements of a theory generated from an axiomatization of the *formal semantics of the RTL language*, *e.g.*, Verilog. Our inference rules aim to provide a simple interface that allows the user to consider simulation at a high-level under any input and state context, and also to manage the complexity of these representations by resolving symbolic expressions to partially concrete ones.

The paper is organized as follows. Section II rigorously defines our strategy construction, including how program semantics are axiomatized in rewriting logic, the definition of the inference rules, and our version of the meta-language. In Section III we provide a number of examples demonstrating the novelty and effectiveness of our framework. A prototype implementation and larger experiments are described in Section IV, and Section V discusses related work and conclusions.

II. A Strategy Language For Stimulus Generation

Our strategy language is formalized within the mathematical framework of *rewriting logic* [10], and takes advantage of the fact that rewriting logic can be used to provide an operational semantics to programming languages [11]. The development involves three rewriting logic specifications, $\mathcal{R}_{RTL}$, $\mathcal{R}_{IR}$, and $\mathcal{R}_{STRAT}$, that build on one another and together define an executable strategy for generating stimuli. $\mathcal{R}_{RTL}$ provides an axiomatization in rewriting logic of the semantics of the RTL language, *e.g.*, Verilog; and proofs in $\mathcal{R}_{RTL}$ naturally correspond to either *concrete* or *symbolic* simulations in the RTL language. Therefore, the stimulus generation problem becomes an exercise in *proof search* from $\mathcal{R}_{RTL}$. To provide a clean interface for controlling this search, we define the theory $\mathcal{R}_{IR}$. $\mathcal{R}_{IR}$ uses reflection to meta-represent $\mathcal{R}_{RTL}$ as a data structure, $\hat{\mathcal{R}}_{RTL}$, and defines a set of *inference rules* specialized for the generation of stimuli. Finally, $\mathcal{R}_{STRAT}$ is a user-defined specification that includes $\mathcal{R}_{IR}$ and orchestrates the application of the inference rules in the search for a computation and stimulus satisfying some desired property.

The theories $\mathcal{R}_{RTL}$, $\mathcal{R}_{IR}$, and $\mathcal{R}_{STRAT}$ all have clear analogies in the ML/LCF setup. To prove a theorem about group theory in LCF, for example, we need to define the set of axioms for groups, say Γ, and then use the inference rules of LCF to establish our theorem, *e.g.*, $\Gamma \vdash x \cdot 1 = x$. The theory $\mathcal{R}_{RTL}$ serves the role of Γ, and $\mathcal{R}_{IR}$ corresponds to LCF's inference rules. Finally, the same way that ML is the functional framework in which an LCF tactic is written, the Maude rewriting logic language provides the logical framework in which the theory $\mathcal{R}_{STRAT}$ is written.

A. Preliminaries

Rewriting logic [10] is a computational logic for expressing concurrency. A rewriting logic *specification*, $\mathcal{R}$, is a triple (Σ, E, R) with Σ a set of function symbols (*e.g.*, $0, 1$ as constants and $+$ as a binary function symbol), E a set of equations characterizing the states of a system as an algebraic data type (*e.g.*, $x + 0 = x$), and R a collection of rewrite rules defining the allowable transitions between equivalence classes of states (*e.g.*, $[x] \longrightarrow [x + 1]$ and $[x] \longrightarrow [0]$ define a counter with reset). More precisely, Σ defines a many-sorted set of terms denoted $T_\Sigma(X)$ with X a denumerable set of variables for each sort; a *ground term* ($t \in T_\Sigma \subset T_\Sigma(X)$) is a term without variables. E contains equations of the form $t = t'$ with $t, t' \in T_\Sigma(X)$ terms of the same sort; and R consists of a set of rewrite rules of the form $t \longrightarrow t'$ *if* C, where, again, t and t' are terms with the same sort, C is a condition, and t' introduces no extra variables. Deduction in rewriting logic establishes judgments of the form $(\Sigma, E, R) \vdash t \longrightarrow t'$, indicating that state t' is *reachable* from state t by rewriting *zero, one, or more rewrite steps* modulo the equations E. It is common to write $\longrightarrow$ as $\longrightarrow^*$ to emphasize this. For a more thorough introduction to rewriting logic, see [10], [12].

B. The Theory $\mathcal{R}_{RTL}$

Rewriting logic provides a good formal basis for stimulus generation because of its ability to cleanly axiomatize the semantics of a wide range of programming languages [11]. For example, Java and Lisp have been given a semantics in this way, and we are working on a rewriting logic semantics for Verilog (see Section IV). For the purposes of this paper we assume that $\mathcal{R}_{RTL} = (\Sigma, E, R)$ provides an operational-style semantics of the form described in [11] for an RTL language of interest. This means that (Σ, E) defines an algebraic data type corresponding to *program configurations*, and that R defines a relation on program configurations that corresponds with *computation steps*. A *configuration* is just a pair of a program, p, and a state, σ, and is written $\langle p, \sigma \rangle$. Judgments in $\mathcal{R}_{RTL}$ are therefore of the form

$$\mathcal{R}_{RTL} \vdash \langle p, \sigma \rangle \longrightarrow \langle p', \sigma' \rangle$$

```
module example(i, clk);
  input        clk;
  input [31:0] i;
  reg   [31:0] x,y;

  always @(posedge clk) begin
    x <= i;
    y <= x + y;
  end
endmodule
```

Fig. 2. Example Verilog Module.

and mean that $\langle p', \sigma' \rangle$ is the result of stepping some number of *computation steps* from $\langle p, \sigma \rangle$. A configuration does not have to be a *ground term*; and when $\langle p, \sigma \rangle$ *actually does* contain symbolic variables, we usually write $\langle p, \sigma(x) \rangle$ to emphasize this fact. Furthermore, judgments involving non-ground terms automatically correspond to *symbolic simulation*; no additional infrastructure is necessary.

Suppose that $\mathcal{R}_{RTL}$ defines a semantics for Verilog. It is beyond the scope of this paper to define this semantics precisely, but we can give a high-level summary. A configuration $\langle p, \sigma \rangle$ contains in p the set of all active processes, and in σ a representation of both the input stream and the current state of all named nodes. As an example, let us consider the Verilog module in Fig. 2 and see what sort of judgments we expect to be able to prove from $\mathcal{R}_{RTL}$. We assume an initial state $\sigma(x)$ defined using the following Maude-like pseudo-code:

```
sigma =  -- initial state of the configuration
 istream: [("i",4)->x][("clk",5)->1]
 store:   ["clk"->0]["i"->0]["x"->0]["y"->0]
```

The input stream maps identifier, future-time pairs to input values, *e.g.*, `[("i",4)->`x`]` means that the identifier `i` will get the symbolic value x in 4 time units (`#4` in Verilog). Similarly, the store maps identifiers to their current values. If $\mathcal{R}_{RTL}$ is defined correctly,

$$\mathcal{R}_{RTL} \vdash \langle p, \sigma(x) \rangle \longrightarrow \langle p', \sigma'(x) \rangle$$

should hold for each of the following $\sigma'(x)$, all of which are reachable in some number of computation steps from the above $\sigma(x)$.

```
sigma' = -- update time 4 units
 istream: [("clk",1)->1]
 store:   ["clk"->0]["i"->x]["x"->0]["y"->0]

sigma' = -- update time 1 unit
 istream: empty
 store:   ["clk"->1]["i"->x]["x"->0]["y"->0]

sigma' = -- process always block
 istream: empty
 store:   ["clk"->1]["i"->x]["x"->x]["y"->0]
```

C. The Theory $\mathcal{R}_{IR}$

As explained above, $\mathcal{R}_{IR}$ provides an interface to search for proofs that discharge our testing goals. Let us, then, define precisely what we mean by a "testing goal". Given a program p and an initial state $\sigma(x)$ with $x \in X$ representing the input stream, we define the *stimulus generation problem* to be the finding of a *ground* substitution $\rho : \{x\} \longrightarrow T_\Sigma$ and a configuration $\langle p', \sigma' \rangle$ such that

$$Q\Big(\mathcal{R}_{RTL} \vdash \rho(\langle p, \sigma(x) \rangle) \longrightarrow \langle p', \sigma' \rangle\Big);$$

where Q is a predicate describing the desired property of the computation, *e.g.*, ρ might instantiate x to a load instruction and Q might insist that $\langle p', \sigma' \rangle$ exhibits a cache load hit. The process through which we *search* for a satisfying substitution ρ and $\langle p', \sigma' \rangle$ is controlled through $\mathcal{R}_{IR}$.

Based on the above definition we limit our consideration to judgments of the form

$$\mathcal{R}_{RTL} \vdash \rho(\langle p, \sigma(\vec{x}) \rangle) \longrightarrow \langle p', \sigma'(\vec{y}) \rangle$$

where $\vec{x}, \vec{y}$ are sets of variables, and $\rho : \vec{x} \longrightarrow T_\Sigma(\vec{y})$ is a substitution. Note that ρ is *not required to be a ground substitution*. As an equivalent but more suggestive form of the above judgment we will write

$$\mathcal{R}_{RTL} \vdash \langle p, \sigma(\vec{x}) \rangle \overset{\rho}{\rightsquigarrow} \langle p', \sigma'(\vec{y}) \rangle$$

The key idea is that we associate *directly* with every judgment a *substitution ρ that encodes the portion of the input stream that has so far been resolved to partially instantiated values*.

The theory $\mathcal{R}_{IR}$ defines a set of four *combinators* for producing judgments of the above form. The typings of these combinators are

```
ID  : Configuration          -> Judgment
T   : Judgment Judgment      -> Judgment
I   : Judgment Substitution -> Judgment
RW  : Judgment               -> Judgment
```

Each combinator corresponds to an inference rule given in Fig. 3. The **ID** rule generates a judgment from an initial configuration and is sound by virtue of the *identity* inference rule from rewriting logic [10]. In $\mathcal{R}_{IR}$ it is defined as

```
ID(< P, SIGMA >) = < P, SIGMA, id, P, SIGMA >
```

where we use matching to get the components `P` and `SIGMA` from the argument configuration, and the resulting judgment is written as a five-tuple. Capital letters are used for variables, and `id` is a constant symbol denoting the *identity* substitution. The **I** rule allows one to further *instantiate* an input stream by composing ρ with another, arbitrary substitution ρ'. This rule follows from the fact that rewriting is *stable under substitution*,

$$\frac{}{t \overset{\mathrm{id}}{\rightsquigarrow} t}\ \textbf{ID} \qquad \frac{t \overset{\rho_1}{\rightsquigarrow} t' \qquad t' \overset{\rho_2}{\rightsquigarrow} t''}{t \overset{\rho_2 \circ \rho_1}{\rightsquigarrow} t''}\ \textbf{T} \qquad \frac{t \overset{\rho}{\rightsquigarrow} t'}{t \overset{\rho' \circ \rho}{\rightsquigarrow} \rho'(t')}\ \textbf{I} \qquad \frac{t \overset{\rho}{\rightsquigarrow} t' \qquad t' \longrightarrow^1 t''}{t \overset{\rho}{\rightsquigarrow} t''}\ \textbf{RW}$$

Fig. 3. Inference Rules for Building Stimulus Generation Strategies.

i.e., $t \longrightarrow t'$ implies $\rho(t) \longrightarrow \rho(t')$ for any substitution ρ. It is defined as

```
compose : Substitution Substitution -> Substitution
compose(RHO1, RHO2) = ... -- compose substitutions

I(<P1, SIGMA1, RHO1, P2, SIGMA2>, RHO2)
  = <P1, SIGMA1, compose(RHO2, RHO1), P2, SIGMA2>
```

The **T** rule allows for two judgments to be combined by composing their component substitutions. It is sound by virtue of *transitivity* in rewriting logic and stability under substitution.

```
T(<P1, SIGMA1, RHO1, P2, SIGMA2>,
  <P2, SIGMA2, RHO2, P3, SIGMA3>)
    = <P1, SIGMA1, compose(RHO2, RHO1), P3, SIGMA3>

T(JMNT1, JMNT2) = JMNT1 [owise]
```

In the above definition we use matching to ensure that the second configuration of the first judgment matches the first configuration of the second judgment, which is required for transitive composition. In order to ensure that `T` is well-defined in all cases, the second part of the definition defaults to returning the first judgment when the two arguments do not match. The **RW** rule allows a judgment to be extended by a *single rewrite* step (computation step). This one step rewrite relation is denoted $\longrightarrow^1$, and $t \longrightarrow^1 t'$ derivations are formed using the inference rules of rewriting logic [10].

```
RW(<P1, SIGMA1, RHO, P2, SIGMA2>) =
  let <P3, SIGMA3> :=
    metaRewrite(oR-RTL, <P2, SIGMA2>, 1)
  in <P1, SIGMA1, RHO, P3, SIGMA3>

RW(JMNT) = JMNT [owise]
```

The constant `oR-RTL` is the object-level representation of $\mathcal{R}_{RTL}$, *i.e.*, it is the meta-representation $\hat{\mathcal{R}}_{RTL}$. The `metaRewrite` function is borrowed from Maude; its takes a meta-represented rewrite theory, a term to rewrite, and a value for the number of steps to rewrite (see [12, §14.4.3]). The reason for the extra default case is that `metaRewrite` is a partial function. It will only produce a result if the term can *actually be rewritten*.

It is worth noting that our inference system *does not* expose the inference rules of rewriting logic directly to the user. This allows for a user-interface that treats simulation only at a high-level, something which seems appropriate for directed testing. Exposing the inference rules of rewriting logic directly to the user would yield a full formal verification environment.

D. The Theories $\mathcal{R}_{STRAT}$: An Example

While $\mathcal{R}_{RTL}$ and $\mathcal{R}_{IR}$ are essentially pre-defined and fixed for the user, recall from our ML/LCF analogy that $\mathcal{R}_{STRAT}$ is fully user-defined, just like an LCF tactic written in ML. In this section we consider writing a strategy $\mathcal{R}_{STRAT}$ for the example module in Fig. 2. As a testing goal, assume that we add to the module a new signal `hit` and a continuous assignment

```
assign hit = x == y && y[1:0] == 0;
```

Of course, the goal is to generate an input sequence that forces `hit` to be 1. Furthermore, let us assume an initial configuration $\sigma(x)$ with the input stream entirely symbolic

```
sigma = -- initial configuration
  istream: x
  store:   ["clk"->0]["i"->0]["x"->0]["y"->1]
           ["hit"->0]
```

Writing a directed test to hit the above goal is not difficult, but this example illustrates a problem similar to the one given in the introduction. Recall that our goal in the introduction was to hit a memory value while it is being pushed out of the cache and up the hierarchy. The main difficulty was that we needed to look into the future state of the simulation in order to plan when we should push a matching load into the instruction stream. Otherwise, by the time we see the cache line eviction it will be too late to start the matching load. The same is true in our example: once we see that `y[1:0] == 0`, it is too late to latch a value into `x`.

Our strategy is very simple. It will generate sequential values for `i`, $1, 2, 3, \ldots$; monitor for the condition `y[1:0] == 0`; and then backtrack and change the previous value of `i` to match the current value of `y`. It can be implemented in just a few lines of Maude code

```
strat(JMNT, V, T) =
  let RHOA  := [x->[("clk", 1+T)->0]
                   [("i"  , 4+T)->V]
                   [("clk", 5+T)->1]x]
      JMNT' := go(I(JMNT,RHOA))
      VALY  := getVal0f("y", JMNT')
      RHOB  := [x->[("clk", 1+T)->0]
                   [("i"  , 4+T)->VALY]
                   [("clk", 5+T)->1]]
  in if   (VALY % 4 == 0)
     then go(I(JMNT, RHOB))
     else strat(JMNT', V+1, T+5)
```

The substitutions `RHOA` and `RHOB` map the variable x to a *partial* concrete input stream that pumps the clock for one cycle and assigns the next value of the input `i`. The remainder of the input stream is delayed by ensuring that the substitution maps x to a non-ground term. The function `go` is assumed to apply the `RW` combinator until all pending *concrete* input stream events are handled and the input stream becomes entirely *symbolic*. `getVal0f("y", JMNT')` is assumed to get the symbolic value currently associated to `y` in the store of

the computation denoted by JMNT'. In the case of the above strategy, the value associated to y will always be concrete, because RHOA and RHOB are constructed such that clk and i are concrete. If we find that the condition on y is met, then we *backtrack* to the previous step and apply RHOB to JMNT using the **I** rule; otherwise we proceed by applying RHOA to JMNT', again via the **I** rule. The strategy gets executed as follows

```
-- call 1: strat(ID(CNFG), 1, 0)
RHOA1 = [x->[("clk" 1)->0][("i",4)->1]
          [("clk",5)->1]x]
-- x <= 1, y <= 1 in JMNT', recurse

-- call 2: strat(JMNT, 2, 5)
RHOA2 = [x->[("clk" 6)->0][("i",9)->2]
          [("clk",10)->1]x]
-- x <= 2, y <= 2 in JMNT', recurse

-- call 3: strat(JMNT, 3, 10)
RHOA3 = [x->[("clk" 11)->0][("i",14)->3]
          [("clk",15)->1]x]
RHOB3 = [x->[("clk" 11)->0][("i",14)->4]
          [("clk",15)->1]]
-- x <= 3, y <= 4 in JMNT', finish

-- final stimulus
compose(RHOB3, RHOA2, RHOA1)
  = [x->
     [("clk" 1) ->0] [("i",4) ->1] [("clk",5) ->1]
     [("clk" 6) ->0] [("i",9) ->2] [("clk",10)->1]
     [("clk" 11)->0] [("i",14)->4] [("clk",15)->1]]
```

III. Example Strategies for Directed Testing

This section gives four more examples demonstrating our strategy language and how it can be used to good effect. The examples highlight novel features of our language that are not possible via constrained randoms or within widely used HVLs such as SystemVerilog. In particular we take advantage of having *computations as first-class objects* and, in addition, exploit the control that our strategy language provides over the *generation and solving of symbolic expressions*. The examples combine the application of a user's high-level design understanding with automated or programmatic methods used to discharge the most tedious aspects of writing a directed test.

A. Example 1: Extended Constraint Solving

An important feature of many HVLs is the ability to the define input constraints used during constrained-random testing; *e.g.*, you can force the source and destination register of the next instruction to be equal. Having computations as first-class, user-controlled objects, as well as the ability to solve symbolic expressions, we can provide an extended form of user input constraints that look some number of clock cycles into the future state of the simulation. Suppose, recalling the microprocessor of Fig. 1, that we want to test the ALU's adder by running it on all combinations of positive and negative operands. Our language allows the user to constrain the register operands of the next instruction so that, for example, one reads a positive value from the register file and the other reads a negative value during the decode stage.

In order to demonstrate how our strategy language enables the solving of internal constraints such as those in the example just described, consider the following snippet of Verilog code

```
always @(posedge clk) begin
   x <= i ^ x;
   y <= x ^ 32'hDEADBEEF;
end
```

The goal is to direct the next value of input i so that when it propagates to y, y takes some chosen value, say 0. Just as the ALU inputs in the DLX pipeline are internal signals that need to be constrained relative to an input event that occurred some number of clock cycles in the past, we can imagine that y may feed into a sub-circuit where 0 is a corner case that needs to be tested. A simple strategy to generate the required stimulus is given below.

```
strat(JMNT) =
  let RHO   := [x->[("clk",0)->0]
                   [("i"  ,1)->x1]
                   [("clk",2)->1]
                   [("clk",3)->0]
                   [("clk",4)->1]]
      JMNT' := go(I(JMNT, RHO))
      PHI   := eq(getValOf("y", JMNT'), 0)
      RHO'  := solve(PHI)
  in I(JMNT, compose(RHO', RHO))
```

Understanding that it takes two cycles for the input i to affect the value at y, the above strategy first defines a substitution which, when applied to an initial configuration with an entirely symbolic input stream, pumps the clock for two cycles, leaving the other input, i, completely symbolic. Pumping the simulation and constructing the symbolic constraint on y yields PHI. For example, if we assume that both variables x and y have value 32'h77777777 in the initial state, the resulting symbolic expression will be (roughly)

```
(x1 ^ 32'h77777777 ^ 32'hDEADBEEF) = 0
```

SAT solving can be used to solve this expression and force i at cycle 0 to be 32'hA9DAC998, as required.

B. Example 2: Backtracking

Having computations as first-class objects also opens up novel opportunities for backtracking-based strategies. A common context where backtracking is useful is when the latency of an operation is difficult to determine. Suppose that one wants to test the register bypassing paths in the DLX pipeline, say from the writeback stage to the execute stage. Due to interlocks, knowing what instruction will be in which pipeline stage every cycle is non-trivial. However, a simple and likely to be effective method for generating a satisfying test is to: (a) generate a random, concrete ALU instruction, (b) simulate until it reaches the execute stage, (c) check the destination register in writeback, and then (d) backtrack to change the original ALU instruction to match, and finally simulate again. Such a strategy is not guaranteed to succeed; for example, if writeback has a store instruction then it does not have a destination register, but it is likely to succeed and requires no SAT solving, so it is efficient.

As a second example, suppose that we decide to add a victim cache to the DLX data cache and that we need a test that causes a victim cache hit. Writing a directed test for this will be difficult, because it depends on the interaction of a set of memory operations and the cache organization. We can use backtracking by simply: (a) generating random memory traffic, (b) monitoring the victim cache for a valid entry, and (c) programmatically backtracking 1, 2, *etc.* cycles and changing the input instruction to a load to the same address. The idea is to programmatically search for the right number of cycles to backtrack, so that the latency of the new load matches with the victim cache entry.

Backtracking is implemented in a strategy simply by storing a symbolic simulation context so that it can be reverted to later. In the caching example above, we would store all of the simulation contexts from 1, 2, *etc.* cycles previous and revert to them iteratively as we search for the correct latency for a load instruction to collide with the victim cache entry. To demonstrate concretely how backtracking can be implemented, consider the following "maze".

```
always @(posege clk) begin
   case (st)
      `S1   : st <= i ? `S2 : `S3;
      `S2   : st <= i ? `S1 : `S4;
      `S3   : st <= i ? `S5 : `S4;
      `S4   : st <= i ? `S5 : `OUT;
      `S5   : st <= i ? `S2 : `S3;
      `OUT  : st <= `OUT;
   endcase
end
```

The goal is to find an input sequence that forces `st` to equal `` `OUT ``; and we can assume that we start in state `` `S1 ``. An efficient strategy first: (a) checks if the current state has been visited and backtracks if yes, (b) tries value 0 on `i` and recurses, if this fails then the strategy tries 1 and recurses. Symbolic simulation contexts are stored as the recursion of `strat` and `strat'` unfolds.

```
-- second argument is ``visited states''
strat(JMNT, S, T) =
 let ST := getValOf("st", JMNT)
 in if   (ST in S) or (ST == OUT)
    then (JMNT, S ST)
    else strat'(JMNT, S ST, T)

strat'(JMNT, S, T) =
 let RHO0 := [x->[("i"  ,T+1 )->0]
                 [("clk",T+5 )->0]
                 [("clk",T+10)->1]x]
     RHO1 := [x->[("i"  ,T+1 )->1]
                 [("clk",T+5 )->0]
                 [("clk",T+10)->1]x]
     (JMNT',S') := strat(go(I(JMNT,RHO0)), T+10)
 in if   (OUT in S')
    then (JMNT',S')
    else strat(go(I(JMNT,RHO1)), S, T+10)
```

C. Example 3: Localized Symbolic Execution

High-level knowledge provided by the user can reduce the complexity of the symbolic expressions sent to SAT. Indeed, simple manipulations to *localize* the symbolic expression provide a balance between user intervention and state space explosion. Suppose that, for example, we want to generate a test for DLX where we have a partial cache load hit, meaning that the load spans two lines with one being in the cache and the other not. Full symbolic simulation may easily result in an intractable problem. However, if the user provides a concrete opcode (*i.e.*, for a load instruction) and a base register, the problem becomes markedly simpler. Now, the symbolic problem just involves finding an appropriate offset value.

We demonstrate how this technique gets expressed as a strategy by considering a simple ALU.

```
always @(posedge clk) begin
   case (op)
      2'b0 : out <= v1 + v2;
      2'b1 : out <= v1 * v2;
      2'b2 : out <= v1 ^ v2;
      2'b3 : out <= v1 == 0;
   endcase
end
```

The goal is to force the output to be 0, and `op`, `v1`, and `v2` are all inputs. To localize the symbolic execution, we will force it to consider just one of the four operations provided by the ALU. This is accomplished by forcing the input `op` to be a concrete value, specifically we give the value 2, meaning that we only consider the exclusive-or operation provided by the ALU. In addition, we will set `v1` to be an arbitrary constant value, 0. The strategy just builds the initial substitution, does symbolic simulation, and solves.

```
strat(JMNT) =
 let RHO   := [x->[("op" ,1 )->2 ]
                  [("v1" ,1 )->0 ]
                  [("v2" ,1 )->x1]
                  [("clk",5 )->0 ]
                  [("clk",10)->1 ]]
     JMNT' := go(I(JMNT,RHO))
     PHI   := eq(getValOf("out", JMNT'), 0)
     RHO'  := solve(PHI)
 in I(JMNT, compose(RHO', RHO))
```

D. Example 4: Another Localized Symbolic Execution

Internal pipeline registers sometimes map very directly to input constraints. For example, in the DLX pipeline the opcodes and source and target registers are carried along from the fetch stage through internal pipeline registers. The pipeline stalls an instruction in the decode stage whenever it is dependent on the instruction in the execute stage *and* that instruction is a memory instruction (load). A user can take advantage of this high-level knowledge by: (a) constructing a configuration with these internal registers having *symbolic variables*, (b) doing symbolic simulation on this configuration, (c) solving the constraint for those variables, and finally (d) mapping the solved values back to an input stream.

As a concrete example, consider the following Verilog snippet.

```
assign e = ((y + z) & 3) == 0;
```

```
always @(posedge clk) begin
 x <= i;
 y <= x;
 z <= y;
end
```

and suppose that we want an input stream that eventually witnesses `e` get set to 1. Our strategy just follows the steps (a) – (d) above, so that we first create an initial configuration where `y` and `z` are given symbolic values y and z; followed by doing symbolic simulation and solving for y and z in the symbolic expression for e; this portion is accomplished as follows

```
stepsAToC =
  let SIGMA := istream: [("clk",5)->0][("clk",10)->1]
               store:   ["i"->0]["e"->0]
                        ["x"->x]
                        ["y"->y]
                        ["z"->z]
      JMNT  := go(ID(<prgm,SIGMA>))
      PHI   := getValOf("e", JMNT)
  in solve(PHI)
```

Finally, with the resulting substitution, say ρ, we would simply use $\rho(z)$ as the first cycle's input, and we use $\rho(y)$ as the second cycle's input.

IV. Tool Implementation and Experiments

We have implemented a prototype of our strategy language using the Maude system [12], which consists of both a language for defining rewriting logic specifications and an interpreter for executing these specifications. This section reports on two larger experiments using this tool. The corresponding $\mathcal{R}_{RTL}$ was developed for Verilog; and we employ the SMT solver STP [13] to resolve queries to `solve`. Although we do not have space enough for details, we emphasize that $\mathcal{R}_{RTL}$ is not limited to just the so-called *synthesizable* subset of Verilog, as other tools are, and exposes all concurrency/non-determinism in the language. It is also worth noting that queries to STP occur outside the otherwise purely rewriting logic formulation. This is simply a matter of pragmatics as solvers such as STP are highly optimized and efficient at discharging our bit-vector-based queries. At present our prototype is rather cumbersome to use, *e.g.*, interfacing with STP is done manually; however, based on our experience building the prototype we are now developing a much more substantial implementation which is faster, more robust, and is easy to use.

Our first experiment is based on the implementation from [14] of an Internet Protocol (IP) lookup. The hardware receives input packets and uses the header information to determine where they should be routed. To avoid massive on-chip memories, a hierarchical table is implemented in [14] that trades latency of individual lookups for a smaller memory size. The latency of an IP lookup roughly amounts to the number of memory reads needed to resolve it, and so any given lookup can be re-circulated through the system anywhere from once to some number of times bounded by a maximum threshold, say t. A reasonable set of coverage goals is to find lookups exercising each of the $1 - t$ recirculation cases. Considering just the case where 3 memory reads are needed, our first strategy applied full symbolic simulation for 10 cycles and then queried STP. This experiment yielded $28,104$ lines of input to STP and required 6.5 seconds to solve on an Intel Core 2 Duo, 2.4GHz with 2GB RAM. Although 6.5 seconds is reasonably fast, we can perform a small modification to our strategy that generates the test much more quickly. Instead of doing full symbolic simulation we instead keep just the first cycle's input symbolic, but for cycles $2 - 9$ we provide arbitrary concrete lookups. When we do this we generate a symbolic problem that is $8,737$ lines of STP input and requires just 0.5 seconds to solve. Both strategies return concrete stimulus that hits the coverage goal. Note that high-level knowledge about the design, *i.e.*, that it is *one-input-one-output*, was required to make this simplification and understand why it should succeed.

Our second example is based on the sort of architecture and challenges one expects in a microprocessor. The Verilog is split into three modules, $u_1 - u_3$, that are strung together to form a pipeline; the pipeline takes one 42-bit input each cycle that is comprised of a 10-bit "opcode" and a 32-bit data payload. u_1 models the front-end of a microprocessor as a high flop count device with regular, but non-fixed latency that fetches and buffers instructions before staging them for execution. The update function for this state is very complicated and depends on most of the microprocessor. We have modeled u_1 via a large RAM that gets updated randomly each cycle and use the data from the RAM to determine a small skew in pushing the inputs to u_2, which models a scheduler. The scheduler is mainly distinguished from the front-end by having a less predictable latency for any given instruction. u_2 is therefore comprised of a buffer that gets sorted each cycle based on the contents of the RAM from u_1. The keys for sorting are given by the 10-bit opcode of each input. Execution, u_3, corresponds to arithmetic or a cache lookup, and we have modeled it as a relatively simple Boolean function of a 32-bit register which is updated with a new random value every cycle. The whole design is about 200 lines of Verilog, but uses loops, arrays, negative clock edges, and other behavioral constructs.

Consider a coverage goal that is mostly concerned with execution and that is only somewhat sensitive to latency, *e.g.*, a cache load hit mostly depends on the address of the load instruction rather than on its latency in reaching the cache. However, most of the logic that gets used before execution is determining the *latency* of the load. Therefore, what we want in a strategy is a way of disregarding the latency aspect and instead use the solver's power just to resolve the address for us. If we just use symbolic simulation on our design for 10 cycles, the result is a $72,795$ input file for STP requiring over two hours to solve, after which we let it time-out. Now, in a real system a more sophisticated mechanism would be needed, but in our case we can use high-level knowledge of the design to create a strategy that distinguishes the top 10-bits, used to control latency, from the bottom 32-bits, corresponding with address. That is, we make the upper 10-bits some arbitrary value, and leave the lower 32-bits symbolic. This simple

strategy produces a symbolic problem that has 10,028 lines of input, is solved in 10 seconds, and yields exactly the stimulus we need. That is, a sequence of 42-bit values such that the first one output from u_3 has an address matching that cycle's randomly determined "coverage value".

V. Related Work and Conclusions

Recent improvements in simulation-based RTL verification are largely concerned with pushing forward the constrained-random paradigm *e.g.*, see [1]–[4]. These advances have been crucial in making constrained-randoms the most important, effective method through which functional verification now takes place. Far less has been done toward improving directed testing methods, and as far as we know our work is the first to specifically address the deficiency in *languages* for crafting directed tests. Ho *et al.* [5] describe an automatic method for generating targeted stimulus which has been very successful, but due to capacity limits of the underlying formal engines it cannot be relied upon to discharge all of the testing obligations after constrained-randoms level off. Software-oriented testing methods such as [15], [16] have also been very successful, but the primary mechanism through which they work, enumerating path constraints, does not appear to be easily applied to hardware and languages such as Verilog. Proof-based verification methods are widely used for certain classes of problems, *e.g.*, see [17], but have not been able to unseat testing from its preeminent position. The *reFLect* [9] language utilizes reflection extensively, including to reason about embedded RTL languages. However, the intended application of *reFLect* is primarily theorem-proving. Finally, we note that strategies of a different sort occur widely in rewriting/program evaluation, although these are used for a very different purpose. A general strategy language that can be used to control rewriting in Maude is defined in [18].

Automated methods handle much of the work in RTL logic verification, but what these methods do not handle can turn into a bottleneck when engineers lack the tools to discharge testing obligations themselves. Therefore, improving the way engineers create directed tests can have a direct impact on the time it takes to finish verifying a design. In this paper we presented a novel language to aid in the creation of directed tests for RTL logic verification, mitigating many tedious aspects currently involved in creating such tests. Our language makes symbolic simulation and the input stimulus context a first-class object and supports advanced constraint solving mechanisms such as SAT and SMT solving. We have demonstrated a number of directed testing examples where these features are useful. In the future we plan to build an easier to use tool, do more extensive case studies, and look at other strategy language features that may be helpful for creating directed tests. We are also interested in the possibility of an interactive symbolic simulator for this task.

Acknowledgments

We gratefully acknowledge the help of Sean Keller, Elliott Fleming and Nirav Dave, all of whom provided detailed comments and discussion on earlier drafts. We are also thankful for discussions with Vijay Ganesh and for his help with STP.

References

[1] H. Kim, H. Jin, K. Ravi, P. Spacek, J. Pierce, B. Kurshan, and F. Somenzi, "Application of Formal Word-Level Analysis to Constrained Random Simulation," in *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 487–490.

[2] N. Kitchen and A. Kuehlmann, "Stimulus Generation for Constrained Random Simulation," in *ICCAD 2007. IEEE/ACM International Conference on Computer-Aided Design*, Nov. 2007, pp. 258–265.

[3] I. Wagner, V. Bertacco, and T. Austin, "Microprocessor Verification via Feedback-Adjusted Markov Models," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 6, pp. 1126–1138, June 2007.

[4] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, "Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification," *Design & Test of Computers, IEEE*, vol. 21, no. 2, pp. 84–93, Mar-Apr 2004.

[5] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart Simulation Using Collaborative Formal and Simulation Engines," in *ICCAD '00: Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*. Piscataway, NJ, USA: IEEE Press, 2000, pp. 120–126.

[6] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[7] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth, "A Metalanguage for Interactive Proof in LCF," in *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA: ACM, 1978, pp. 119–130.

[8] M. Kaufmann, J. S. Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[9] J. Grundy, T. F. Melham, and J. W. O'Leary, "A reflective functional language for hardware design and theorem proving," *J. Funct. Program.*, vol. 16, no. 2, pp. 157–196, 2006.

[10] J. Meseguer, "Conditional Rewriting Logic as a Unified Model of Concurrency," *Theor. Comput. Sci.*, vol. 96, no. 1, pp. 73–155, 1992.

[11] J. Meseguer and G. Roşu, "The Rewriting Logic Semantics Project," *Theoretical Computer Science*, vol. 373, no. 3, pp. 213–237, 2007.

[12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude - A High-Performance Logical Framework*, ser. LNCS. Springer, 2007, vol. 4350.

[13] V. Ganesh and D. L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," in *CAV*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 519–531.

[14] Arvind, R. S. Nikhil, D. L. Rosenband, and N. Dave, "High-level Synthesis: An Essential Ingredient for Designing Complex ASICs," in *ICCAD*. IEEE Computer Society / ACM, 2004, pp. 775–782.

[15] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 213–223.

[16] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *OSDI*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224.

[17] D. M. Russinoff, "A Case Study in Fomal Verification of Register-Transfer Logic with ACL2: The Floating Point Adder of the AMD Athlon^{TM} Processor," in *FMCAD*, ser. Lecture Notes in Computer Science, W. A. H. Jr. and S. D. Johnson, Eds., vol. 1954. Springer, 2000, pp. 3–36.

[18] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo, "Deduction, strategies, and rewriting," *Electr. Notes Theor. Comput. Sci.*, vol. 174, no. 11, pp. 3–25, 2007.