Abstracting RTL Designs to the Term Level

Bryan A. Brady
UC Berkeley
bbrady@eecs.berkeley.edu

Randal E. Bryant
Carnegie Mellon University
randy.bryant@cs.cmu.edu

Sanjit A. Seshia
UC Berkeley
sseshia@eecs.berkeley.edu

October 12, 2008

Abstract

Term-level verification is a formal technique that seeks to verify RTL hardware descriptions by abstracting away details of data representations and operations. The key to making term-level verification automatic and efficient is in deciding what to abstract. We investigate this question in this paper and propose a solution based on the use of type qualifiers. First, we demonstrate through case studies that only selective term-level abstraction can be very effective in reducing the run-time of formal tools while still retaining precision of analysis. Second, the term-level abstraction process can be guided using lightweight type qualifiers. We present an annotation language and type inference scheme that is applied to the formal verification of the Verilog implementation of a chip multiprocessor router. Experimental results indicate type-based selective term-level abstraction is effective at scaling up verification with minimal designer guidance.

1 Introduction

Register-transfer-level (RTL) descriptions are often the most authoritative models of a system. It is therefore essential for formal verification tools to operate at the RTL. Most formal verification tools however, turn the RTL model into a bit-level model upon which a bit-level technique such as finite-state model checking [10] is invoked. Employing techniques such as predicate abstraction on the RTL (e.g., [18]) helps to scale the analysis further for verifying control-dependent properties. However, for proving data-dependent properties including equivalence or refinement checking, bit-level techniques run into state-space explosion and predicate abstraction requires far too many predicates, necessitating additional abstraction.

Term-level modeling seeks to make formal verification of data-intensive properties tractable by abstracting away details of data representations and operations, viewing data as symbolic *terms*. Term-level abstraction has been found to be especially useful in microprocessor design verification, using techniques such as term-level bounded model checking, correspondence checking, refinement verification, and predicate abstraction of term-level models [11, 17, 21, 22]. The precise functionality of operations of units such as instruction decoders and the ALU are abstracted away using *uninterpreted functions*, and decidable fragments of first-order logic are employed in modeling memories, queues, counters, and other common constructs (e.g., as performed in the UCLID verification system [9]). Efficient SAT-based decision procedures for fragments of first-order logic are used as the computational engines for term-level verifiers.

Simply abstracting all Boolean signals to Boolean variables and all bit-vector signals to terms results in too abstract a model, in which properties of bit-wise and finite-precision operators are obscured. Dealing with

this "precision gap" is a significant hurdle to the wider adoption of term-level verification, since manual abstraction is tedious and error-prone. Thus, the central problem in using term-level abstraction is to decide, in a sound manner, *what to abstract*.

Contributions. We propose the use of selective term-level abstraction based on *type qualifiers*, optionally guided by designer-provided annotations in the RTL. Our approach complements the use of fast SAT-based decision procedures for word-level and term-level reasoning. We demonstrate our approach on the RTL description of a chip multiprocessor router. The presented approach is lightweight, requiring very little work on the part of the designer, with very few annotations sufficing to reduce the verification time substantially. The light annotation burden is made possible by use of automated type inference. These rules check that the designer-specified term abstractions can be applied safely, and infer all the signals to which the term abstraction can be propagated. Experimental results indicate that type-based selective term-level abstraction can be very effective at scaling up verification.

Outline. In Section 2, we review background material and related work. We describe the type annotation language in Section 3 along with an automatic technique for transforming annotated Verilog code to a combined word-level (bit-vector) and term-level model for the UCLID verifier [9]. Experimental results on the Verilog design of a chip multiprocessor router [23] demonstrate how our approach can speed up verification.

2 Background and Related Work

Word-level modeling of designs involves representing control signals as Boolean variables, data as bit-vector variables, and memories as arrays of bit-vector variables. Verification of a word-level model can be performed in two ways. The first method is to turn all bit-vector variables into a vector of Boolean variables of the defined size, thus creating an entirely bit-level netlist. Standard finite-state model checking techniques [12] can be used on the resulting netlist.

Recently, however, efficient SAT-based decision procedures have been developed for finite-precision bit-vector arithmetic (e.g., [6, 8, 13, 15]). These procedures reason at a level of abstraction higher than the bit level by the use of word-level simplification rules and abstraction-refinement techniques. Verification methods such as bounded model checking [4] and predicate abstraction [16] are thus performed efficiently at the word level (e.g., [18]). However, since most word-level techniques are ultimately based on bit blasting to SAT, it is often necessary to raise the level of abstraction still higher to speed up verification.

Term-level modeling offers a higher level of abstraction. There are three elements of term-level modeling: *data abstraction*, *function abstraction*, and *memory abstraction*. Data abstraction involves treating bit-vector expressions as abstract terms that are interpreted over a suitable domain (typically a subset of $\mathbb{Z}$). In function abstraction, bit-vector operators and modules computing bit-vector values are treated as "black-box," uninterpreted functions constrained only by functional consistency: that they must evaluate to the same values on the same arguments. Finally, in memory abstraction, memories and data structures are modeled in a suitable theory of arrays or memories, such as by the use of special **read** and **write** functions [11].

In essence, term-level modeling involves representing the system in fragments of first-order logic rather than in propositional logic. The subset of first-order logic commonly used today includes a *combination of three first-order theories:* the theory of equality and uninterpreted functions (EUF), integer linear arithmetic, and a suitable theory of arrays. UCLID [9] is a system for the term-level modeling and verification of systems represented in this subset of first-order logic, with arrays being modeled with restricted lambda expressions.

UCLID now also supports the theory of finite-precision bit-vector arithmetic [8], thus allowing a combination of term-level and word-level modeling, which we exploit in this paper. The verification techniques of particular interest to us in this paper are bounded model checking and checking an implementation design refines its specification. An instance of the latter is correspondence checking of pipelined processors [11].

The Reveal system automatically generates term-level models from Verilog [2]. The underlying logic for term-level modeling in Reveal is CLU, which originally formed the basis for the UCLID system [9], and which is a strict subset of the afore-mentioned combination of theories. Reveal uses a counterexample-guided abstraction-refinement (CEGAR) approach [1, 3]. It starts by completely abstracting a Verilog description to CLU. Next, it attempts to verify correctness of the abstracted design. If the verification succeeds, it terminates. However, if the verification fails, Reveal checks whether the counterexample is spurious using a bit-vector decision procedure. If the counterexample is spurious, a set of bit-vector facts are derived and used on the next iteration of term-level verification. If not, the system terminates, having found a bug.

The CEGAR approach has shown very promising results [2]. In some cases, however, several abstraction-refinement iterations are needed to infer fairly straightforward properties of data, thus imposing a significant overhead. For instance, in one of our examples, a chip multiprocessor router [23], the header field of a packet must be extracted and compared several times to determine whether the packet is correctly forwarded. If any one of these extractions is not modeled precisely at the word-level, a spurious counterexample results. Thus, the translation is complicated by the need to instantiate relations between individually accessed bit fields of a word modeled as a term using special uninterpreted functions to represent concatenation and extraction operations.

Our approach to term-level abstraction is distinct from that of Andraus and Sakallah, yet complementary. We propose the use of type annotations provided by the designer to guide the creation of an initial sound abstraction. If the designer-provided annotations are incorrect, a type checker emits warnings and avoids performing the specified abstractions. A CEGAR approach can then be employed on the output of the proposed type-based approach.

Johannesen presented an automated bitwidth reduction technique which was used to scale down design sizes for RTL property checking [19, 20]. A static data-flow technique is used to partition the datapath signals based on the usage of the individual bits. Bits that are used in a symmetric way are abstracted to take the same value, while preserving satisfiability. As with the approach we present, one is able to compute a satisfying assignment for the original circuit from a satisfying assignment of the reduced circuit.

More recently, Bjesse presented a technique which is an extension to Johannesen's work [5]. The main difference is that Bjesse includes the partitioning of the initial states, ensures that the current- and next-state partitions correspond, and bitblasts operators such as inequalities which are left untouched in Johannesen's work.

Our work, developed concurrently with and independent of Bjesse's [5], is different from that work and Johannesen's [19,20] in that our abstraction is not limited to merely reducing bitwidths of variables. Instead, we encode the different partitions of the circuit in different logical theories. In this paper, we use two theories: finite-precision bit-vector arithmetic (BV) and the logic of equality with uninterpreted functions (EUF), but the ideas are not restricted to BV and EUF. The use of logical theories allows us to use one of several techniques to encode abstracted variables, not just a single procedure; e.g., we are able to use for EUF the technique of positive equality [7] that encodes some variables into the SAT problem with constant values. Furthermore, neither Johannesen's nor Bjesse's reduction technique has a way to incorporate user insight for abstraction. In our work, we allow the user to use type qualifiers to convey his/her intuition as

to how the circuit should be abstracted. If the computed abstraction does not coincide with the designer's intuition, our tool will warn the user and generate suitable feedback. The output from our tool can guide the user to write annotations or rewrite the RTL in a way that substantially reduces verification effort. As we will see with an example in this paper, sometimes a small rewrite of the original code can achieve a large reduction in verification time.

3 Type Qualifiers for Term-Level Abstraction

We propose the use of *type qualifiers* to guide the process of abstracting to the term-level. Type qualifiers are widely used in the static analysis of software [14], from finding security vulnerabilities to the detection of races in concurrent software, to enable programmers to easily guide the analysis. Even standard languages such as C and Java have keywords such as "const" that qualify standard types such as `int`s.

In our case, type qualifiers indicate what can and cannot be abstracted to a fragment of first-order logic supported by the term-level verifier. Their widespread effectiveness in software leads us to believe that they would be effective in incorporating a designer's insights for abstraction, without placing an undue burden on the designer. The simplest form of type qualifier specifies *data abstraction*. In data abstraction, we specify that part of a bit-vector signal must be treated as an uninterpreted `term`. The designer can also indicate that a module is to be treated as `uninterpreted`, which is *function abstraction*. Finally, memories and data structures such as queues can also be abstracted by use of UCLID-style lambda expressions [9].

Our approach is depicted in Figure 1. We start with Verilog RTL, the specification to be verified, and, option-

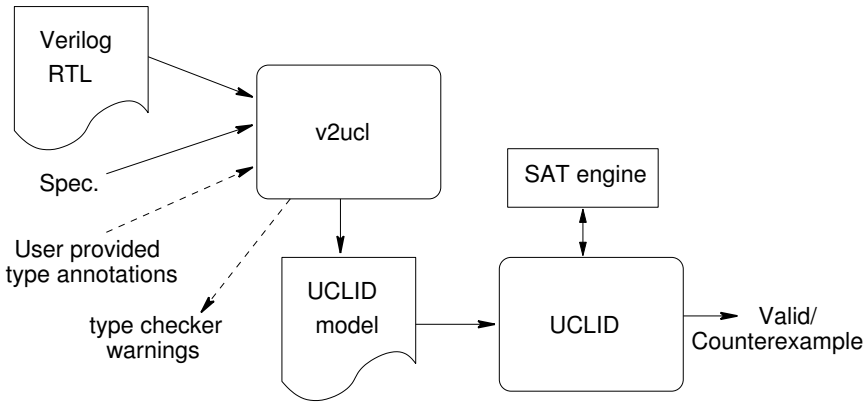

Figure 1: **Overview of Term-Level Abstraction.** The dashed lines indicate optional input and output.

ally, with designer-provided type qualifier annotations. The Verilog-to-UCLID abstraction tool V2UCL then performs type inference and type checking. If the designer-provided annotations are inconsistent, warnings are generated, which indicate to the designer what she needs to fix. Otherwise, V2UCL automatically creates a hybrid term and bit-vector UCLID model, with the specification included in it. The designer can now run UCLID (which uses off-the-shelf SAT engines) and evaluate whether the design satisfies its specification.

The type qualifiers are currently specified in Verilog inside comments, similar to javadoc notation. Rather than using a formal notation for the type qualifiers, we will use this the comment-style notation throughout the paper.

We next describe below the three forms of abstraction and how they are implemented in V2UCL. We illustrate the different kinds of abstraction with code snippets from the Verilog description of a chip multiprocessor (CMP) router design. We begin with a quick overview of this design.

3.1 CMP Router

The chip multiprocessor router design [23] is part of the on-chip interconnection network that connects processor cores with memory and with each other. The design comprises over 1000 lines of Verilog. The original router design has five input ports and five output ports, which we reduced to two input/output ports for this case study. The router's function is to direct incoming packets to the correct output port. Each packet is made up of smaller components, called *flits*. There are three kinds of flits: a *head flit*, which reserves an output port, one or more *body flits*, that contain the data payload, and a *tail flit*, which signals the end of the packet. Figure 2 depicts the structure of a flit: the lower 8 bits (the header) determine the type of flit and its destination, while the top 24 bits store data.

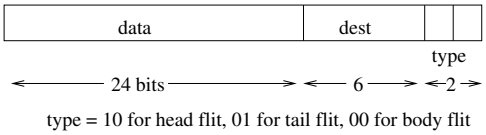

Figure 2: **Anatomy of a flit.** Each flit is 32 bits long.

There are four main modules of the router, as shown in Figure 3. The first module, called the *input controller*,

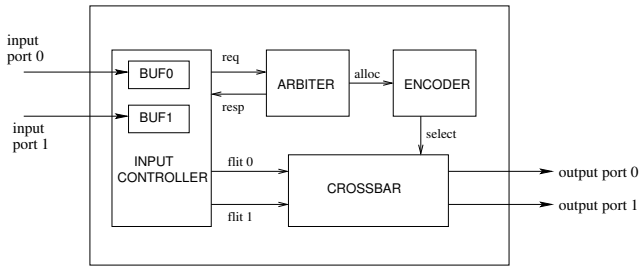

Figure 3: **Chip multiprocessor router block diagram.** There are four main modules: the input controller, the arbiter, the encoder, and the crossbar.

buffers incoming flits, determines their destination port, and interacts with an *arbiter* module in order to reserve an output port. In Peh's design, the reservation of an output port is performed on receipt of a head flit. Thereafter, all body flits and tail flits are directed to the output port without incurring any further latency. The arbiter is fair, assigning priorities to input ports based on a simple round-robin scheme. The remaining modules are the *encoder* and *crossbar*, which contain logic to copy flits to the output port from the input port that has been assigned that output port.

We modified the crossbar implementation for this case study. The original implementation forwarded every bit of a flit individually, thereby eliminating any scope for performing data abstraction. We rewrote it to a word-level crossbar in a few minutes and then checked its equivalence with the bit-level version using the SMV model checker. We believe that this rewriting can be automated and will return to discuss this point in Section 5.

3.2 Data Abstraction

The term abstraction of a bit-vector Verilog signal `foo` of width w is specified as follows:

```
/*@term [w_1 : w'_1] [w_2 : w'_2] ...[w_k : w'_k] */
    <stype> [w − 1:0] foo;
```

where `<stype>` is `input`, `output`, `wire`, or `reg`, and the ranges `[`$w_1 : w'_1$`], [`$w_2 : w'_2$`], ..., [`$w_k : w'_k$`]` are non-overlapping sub-ranges of `[`$w - 1$`:0]`.
In our experience, it is often the case that $k = 1$, meaning that either the entire word is abstracted as a term, or a single part of the word is abstracted.

The `/*@term ...*/` qualifier is always associated with an underlying logic of term-level abstraction. Thus, the `term` qualifier associates a set of allowed (legal) operations with each abstracted term, where that set of operations is the set of function symbols in the associated logic.

For our current implementation, the `/*@term ...*/` qualifier specifies that the abstracted term is in *the logic of equality and uninterpreted functions* (EUF). In other words, the only legal operations on that term are copying, tests for equality or disequality, or as arguments to uninterpreted functions. This is not an inherent restriction of the type qualifier approach; we plan to consider other fragments of first-order logic in the future.

Given the Verilog RTL $\mathcal{M}$, optionally annotated with type qualifiers as given above, and a specification $\mathcal{S}$, V2UCL performs an automatic analysis in the following steps:

1. *Compute Equivalence Classes:* Partition the set of signals and their extracted portions into equivalence classes such that signals that appear together in assignments, relational comparisons, or functional operations in $\mathcal{M}$ or $\mathcal{S}$ end up in the same equivalence class.

2. *Compute Maximal Term Abstraction:* For each equivalence class, we compute the maximal term abstraction (defined in Section 3.2.2) that is common to all signals in that equivalence class.

3. *Check Type Annotations for Consistency:* If the designer has provided any type annotations, we check those for consistency with the computed abstraction. If the designer's annotations are more abstract, an error message is generated. If they are less abstract, the tool uses the automatically inferred abstraction, unless the bit-vector portion to be extracted is too small (currently heuristically set to 4 bits).

4. *Create UCLID Model:* Signals that have associated term abstractions are encoded in the UCLID model with combinations of term and bit-vector variables, and a hybrid UCLID model is generated and verified.

We describe each of these stages in more detail in Sections 3.2.1-3.2.4 below.

3.2.1 Compute Equivalence Classes

This step is performed by computing equivalence classes of bit-vector expressions defined in the RTL with the goal of giving signals in the same equivalence class the same term abstraction.

We first pre-process the input file to identify all bit-vector expressions appearing in the Verilog. Each of these will be termed a *node*. Some nodes are signals.

Then the process of constructing equivalence classes begins.

First, each bit-vector node is placed in its own singleton equivalence class. Denote the equivalence class containing v_i by $\mathcal{E}(v_i)$.

Next, equivalence classes are merged according to the following rules, applied to expressions in both the Verilog RTL $\mathcal{M}$ and the specification $\mathcal{S}$ to be verified:

1. *Assignment:* If v_i is assigned the value of v_j in a combinational or sequential assignment, then we merge $\mathcal{E}(v_i)$ and $\mathcal{E}(v_j)$.
   If v_i is assigned the value of v_j or v_k conditioned on the value of a Boolean signal, then we merge classes $\mathcal{E}(v_i)$, $\mathcal{E}(v_j)$, and $\mathcal{E}(v_k)$.
2. *Relational Comparison:* If v_i and v_j are compared by a relational operator for equality or relative ordering, then we merge $\mathcal{E}(v_i)$ and $\mathcal{E}(v_j)$.
3. *Other Operations:* If v_{i_1}, v_{i_2}, ..., v_{i_n} appear as operands to the same word-level arithmetic or bit-wise operator, we merge the equivalence classes $\mathcal{E}(v_{i_j})$ for $1 \leq j \leq n$.

Note that it is important to include the specification $\mathcal{S}$ also in the analysis, since, e.g., arithmetic operations might be performed on a node in the $\mathcal{S}$ even if it is not performed in the RTL.

3.2.2 Compute Maximal Term Abstraction

For each equivalence class, we compute the *maximal term abstraction* common to all signals in that equivalence class. The maximal term abstraction for a signal v with range $[w-1:0]$ is a sequence of non-overlapping sub-intervals $[w_1 : w_1']$, $[w_2 : w_2']$, ..., $[w_k : w_k']$ of $[w-1:0]$ such that the fanouts of those extractions of v do not appear in any bit-vector arithmetic operation, but those of the extractions $[w, w_1]$, $[w_1', w_2]$, ..., $[w_k', 0]$ do (and hence the latter intervals cannot be abstracted to terms).

The first step is to filter out equivalence classes that cannot be abstracted. For each equivalence class, if any element is used with a bit-vector operator excluding extraction or concatenation, or is compared with a bit-vector constant, or compared with non-equality relational comparison (such as $<$), we will not abstract any signals in that equivalence class, since that operation cannot be modeled in equality logic.

At this point, the only equivalence classes remaining will contain nodes whose fanouts go into assignments, conditional assignments, equality (equals, not equals), or extraction/concatenation operators. If any of these classes have signals of different bit-width, we remove that class from further consideration.

The next step will determine if a consistent term abstraction exists for each remaining equivalence class. The main challenge is in dealing with extractions and concatenations.

Let us first consider concatenations. Our approach is to treat concatenations like extractions, by tracking the sub-intervals that make up the result of a concatenation. For example, if we had the assignment `y = {y1,y2}`, with `y1` being 8 bits and `y2` being 4 bits, we treat `y` as if it had ranges [11:4] and [3:0] extracted from it.

Next we compute how extractions refine the bit-range of each signal.

For each equivalence class $\mathcal{E}$, let the bit-width of any signal in that class be w. For each bit-vector signal in $\mathcal{E}$, v_j, let I_j be the set of all ranges extracted from v_i. Thus, I_j is a set of sub-intervals of the range

$[w-1:0]$.

We take the intersection of all sub-intervals in all sets I_j. That results in a maximally refined partition $R_{\mathcal{E}}$ of the interval $[w-1:0]$.

If the transitive fanout of any extraction in I_j appears in a non-equality comparsion or a bit-vector arithmetic operation (other than concatenation), we remove all intervals contained in it from $R_{\mathcal{E}}$.

The remaining intervals in $R_{\mathcal{E}}$ are guaranteed to appear only under equality logic operations. These intervals then make up the maximal term-level abstraction for class $\mathcal{E}$.

3.2.3 Check Type Annotations for Consistency

Finally, we check designer-provided type annotations against the computed maximal term-level abstraction.

If the intervals that make up the designer's term-level abstraction are contained within intervals of the computed maximal abstraction, the computed abstraction is more abstract, so the tool uses it.

However, if some interval in the designer provided annotation overlaps partially with an interval in the computed maximal abstraction, an error message is generated. The error message indicates the problematic annotation along with the extracted portion of the signal that is inconsistent with the annotation.

3.2.4 Create UCLID Model

Signals that have associated term abstractions are encoded in the UCLID model in a straightforward way.

Suppose that the inferred type qualifier for signal `foo`, corresponding to the maximal term abstraction at the end of the above steps is:

```
/*@term [w_1 : w'_1] [w_2 : w'_2] ...[w_k : w'_k] */
    <stype> [w-1:0] foo;
```

The Verilog signal `foo` is transformed into at most $2k+1$ UCLID signals, of which k are `TERM` valued and the rest remain as bit-vector with the number and widths of these bit-vector variables determined by the quantities $w, w_1, w'_1, \ldots, w_k, w'_k$.

Equalities between bit-vector variables v_i and v_j are transformed into a conjunction of equalities between their corresponding sub-ranges, some of which might be abstracted to terms and others to bit-vectors.

The following code snippet from the CMP router illustrates the data abstraction algorithm. The `...` indicates code irrelevant to the discussion herein.

```
input [31:0] flitin;
...
wire [2:0] destx,desty;
...
assign desty = flitin[4:2];
assign destx = flitin[7:5];
...
assign  vc_req_feed = (~renable) ? 5'b00000 :
(currx < destx) ? 5'b00100 :
(currx > destx) ? 5'b01000 :
```

```
(curry < desty) ? 5'b00001 :
(curry > desty) ? 5'b00010 :
5'b10000;
...
assign enable = (flitin[1:0] == 2'b10);
```

The extracted portions of signal `flitin` are `flitin[1:0]`, `flitin[4:2]`, and `flitin[7:5]`. The relevant equivalence classes are {`flitin`}, {`flitin[1:0]`}, {`flitin[7:5], destx, currx`}, and {`flitin[4:2], desty, curry`}. Since `destx`, `desty`, `currx`, and `curry` all appear under $<$ and $>$ comparisons, and `flitin[1:0]` is compared to a bit-vector constant, none of the latter three equivalence classes can be term-abstracted.

Consider $\mathcal{E} = \{\texttt{flitin}\}$. Then, $I_j = \{[7:5], [4:2], [1:0]\}$. None of these intervals in I_j intersect with each other, so the maximally refined set $R_{\mathcal{E}} = \{[31:8], [7:5], [4:2], [1:0]\}$. The last three elements of this set cannot be abstracted (as reasoned above), so the only remaining interval in $R_{\mathcal{E}}$ is $[31:8]$.

Thus, the output maximal term abstraction of signal `flitin` is `/*@term [31:8] */`.

3.3 Function Abstraction

The abstraction of functional blocks by V2UCL is at present rather rudimentary.

Given a module, the designer can annotate the module declaration with the annotation `/*@uninterpreted */` to indicate to the translator to abstract away the internal logic in the module and simply treat it as a *bit-vector uninterpreted function*, which maps the input bit and bit-vector types to the output bit or bit-vector type.

Note that this treats all module instantiations in a uniform way. An alternative would be to abstract specific instantiations while leaving others unabstracted, but we have not yet explored this option.

Note that treating a module as an uninterpreted function does not imply that the bit-vector inputs or outputs of the module are abstracted as `term`s. Separate annotations will be need to be inferred to perform additional data abstraction.

3.4 Memory Abstraction

Array-typed signals representing memories, queues, and similar data structures are modeled in UCLID using a restricted class of lambda expressions, as described in Section 2. The main challenge in automatic translation of such signals from Verilog to UCLID is in handling the initialization. For example, the array of input buffers is initialized and updated in Verilog as follows:

```
if (reset)
  begin
    for (i=0; i<'NUMBUFFERS; i=i+1)
      buffers[i] <= 'NAF;
  end
else
  begin
    /* updates to head and tail of queue */
```

```
    buffers[head] <= nxt_buffers_head;
    buffers[tail] <= nxt_buffers_tail;
  end
```

The corresponding lambda notation in UCLID is compact in its initialization (the signal `reset` is the initialization signal for the whole router design). The logic for `nxt_buffers_head` and `nxt_buffers_tail` take care of the cases where the buffer is full or empty.

```
init[buffers] := Lambda(i). NAF;
next[buffers] := Lambda(i).
  case
    i = head : nxt_buffers_head;
    i = tail : nxt_buffers_tail;
    default  : buffers(i);
  esac;
```

If a Verilog array typed signal has a single initialization value (such as `buffers` above), the designer indicates it by annotating that signal's declaration with that initialization value, facilitating an easy translation to UCLID.

4 Case Study: CMP Router

We present experimental results with the CMP router design described earlier.[1]

Two UCLID models were created from the CMP router design.

The first model is a purely bit-vector model. The only abstraction performed in this translation is the modeling of input buffers as UCLID-style queues based on lambda expressions [9]; apart from this, the translation process is a straightforward transliteration of Verilog to a UCLID model with purely Boolean and bit-vector datatypes.

The environment of the router was manually modeled in UCLID. The environment injects one packet onto each input port, with the destination of the packets modeled by an uninitialized (symbolic) destination field in the respective head flit; this models scenarios of having packets destined for different output ports as well as for the same output port (resulting in contention to be resolved by the arbiter). Bounded model checking (BMC) was used to check that starting from a reset state, the router correctly forwards both packets to their respective output ports within a fixed number of cycles that depends on the length of the packet. The packet with higher priority has its flits directed to its output port from cycle 7 onward. If both packets are headed for the same output ports, the lower priority packet must wait until all flits of the other packet have been copied to the output. This property was verified successfully by UCLID using purely bit-vector reasoning. Note that a purely term-level model of the router generates spurious counterexamples as the correctness depends on routing logic that performs bit extractions and comparisons on the flit header.

The second model was a hybrid bit-vector and term UCLID model. To generate this model, V2UCL was able to automatically abstract every signal that stores a flit using the term abstraction `/*@term [31:8] */`. No designer guidance was needed.

[1] The abstraction tool and experimental data presented here is made available at http://www.eecs.berkeley.edu/~bbrady/v2ucl .

The times to generate both types of UCLID models from Verilog are within a second (approximately the same).

Experimental Results. We ran experiments with both models for increasing packet size. (Recall that a packet comprises several flits.) For the bit-vector model, UCLID uses its built-in bit-vector decision procedure. For the hybrid model, UCLID reasons in equality logic for the term-abstracted signals and with bit-vector arithmetic for the rest. There were no uninterpreted functions in this model, but buffers were modeled using lambda expressions.

Figure 4 compares the verification run-times for the bit-vector model with those for the hybrid model. We plot two curves for each type of model: one for the time taken by the SAT solver (MiniSat), and another for the time taken by UCLID to generate the SAT problem. We can see that, for the pure bit-vector model,

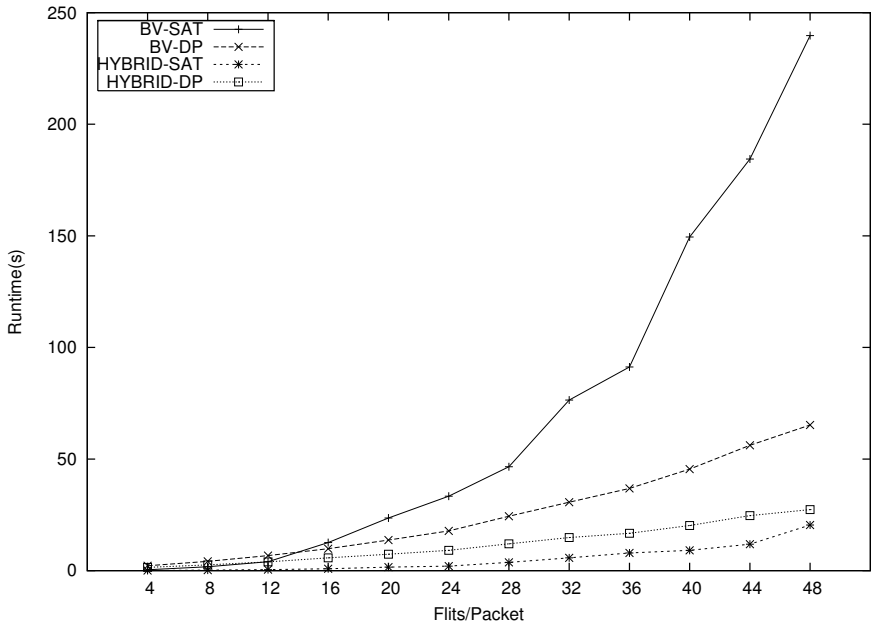

Figure 4: **Comparing run times of hybrid and pure bit-vector models for increasing packet size.** The time for the SAT solver is indicated by "SAT" and the time to generate the SAT problem by "DP."

the time taken by the SAT solver scales exponentially with packet size, whereas that for the hybrid model increases much more gradually. Using the hybrid version achieves a speed-up of about 16X in SAT time. The improvement in time to encode to SAT is more modest.

To evaluate whether the improvement was entirely due to reduction in SAT problem size, we plotted the ratio of speedup in SAT time along with the ratio of reduction in problem size. These plots are shown in Figure 5. We can see that the speedup in time does not track the reduction in problem size (which is largely constant for increasing packet size), indicating that the abstraction is assisting the SAT engine in other ways.

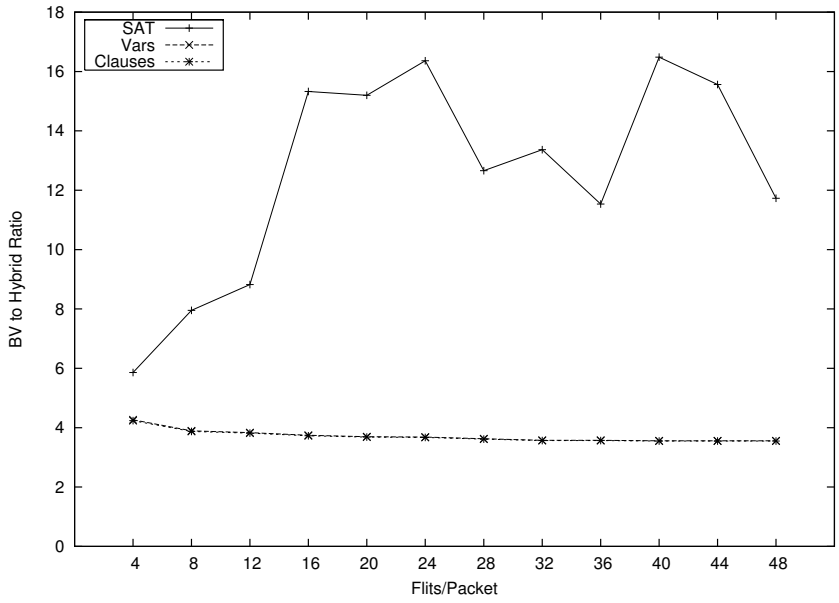

Figure 5: **Comparing reduction in run-time (denoted "SAT") vs. reduction in SAT problem size ("Vars/Clauses"), going from pure bit-vector to hybrid models, for increasing packet size.** The curves for number of variables and number of clauses coincide.

5 Conclusions

We conclude from our experiments that selected term-level abstraction on the original model can greatly reduce verification time. Designers can guide such term-level abstraction through type annotations in the RTL, which, along with type checking, provides an automatic, sound technique to generate a selectively term-abstracted model.

As mentioned earlier in the paper, we found it necessary to manually rewrite a bit-level crossbar in the router Verilog to the word level. The original version was implemented entirely using bit-level multiplexors that copy a 32-bit input packet to the output, where each data input to a multiplexor was a single bit of a packet. In rewriting this module at the word-level, the main insight used was that the select inputs to all multiplexors were the same. We believe that some such "lifting" of designs from the bit-level to the bit-vector level can be automated. For example, the fact that the control inputs (such as the select inputs) to all bit-level modules are the same can indicate that those modules can be merged into a word-level implementation without sacrificing soundness.

For future work, it will also be interesting to combine our type-based approach with a CEGAR approach.

References

[1] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Proceedings of ASP-DAC*, pages 19–24, 2006.

[2] Z. S. Andraus, M. H. Liffiton, and K. A. Sakallah. CEGAR-based formal hardware verification: A case study. Technical Report CSE-TR-531-07, University of Michigan, May 2007.

[3] Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of Verilog models. In *Proceedings of the 41st Design Automation Conference (DAC)*, pages 218–223, 2004.

[4] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference (DAC)*, 1999.

[5] P. Bjesse. A practical approach to word level model checking of industrial netlists. In A. Gupta and S. Malik, editors, *Computer-Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 446–458. Sprinter-Verlag, 2008.

[6] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A lazy and layered SMT(BV) solver for hard industrial verification problems. In *Computer Aided Verification (CAV)*, LNCS 4590, pages 547–560, 2007.

[7] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, January 2001.

[8] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4424 of *Lecture Notes in Computer Science*, pages 358–372. Springer, 2007.

[9] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. G. Larsen, editors, *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.

[10] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *28th Design Automation Conference*, 1991.

[11] J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In D. L. Dill, editor, *Computer-Aided Verification (CAV '94)*, LNCS 818, pages 68–80. Springer-Verlag, June 1994.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[13] B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In *Computer-Aided Verification (CAV)*, pages 296–300, 2005.

[14] J. S. Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, 2002.

[15] V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV)*, LNCS 4590, pages 519–531, 2007.

[16] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[17] W. A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[18] H. Jain, D. Kroening, N. Sharygina, and E. Clarke. Word level predicate abstraction and refinement for verifying RTL Verilog. In *Proceedings of the 42nd Design Automation Conference (DAC)*, pages 445–450, 2005.

[19] P. Johannesen. BOOSTER: Speeding up RTL property checking of digital designs through word-level abstraction. In *Computer Aided Verification*, 2001.

[20] P. Johannesen. *Speeding up hardware verification by automated data path scaling*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2002.

[21] S. K. Lahiri and R. E. Bryant. Deductive verification of advanced out-of-order microprocessors. In *Proc. 15th International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 341–354, 2003.

[22] P. Manolios and S. K. Srinivasan. Refinement maps for efficient verification of processor models. In *Design, Automation, and Test in Europe (DATE)*, pages 1304–1309, 2005.

[23] L.-S. Peh. *Flow Control and Micro-Architectural Mechanisms for Extending the Performance of Interconnection Networks*. PhD thesis, Stanford University, August 2001.