Formalization of the DE2 Language

Warren A. Hunt, Jr. and Erik Reeber

Department of Computer Sciences,
1 University Station, M/S C0500
The University of Texas
Austin, TX 78712-0233, USA
E-mail: {hunt,reeber}@cs.utexas.edu

Abstract. We formalized the **DE2** hierarchical, occurrence-oriented finite state machine (FSM) language, and have developed a proof theory allowing the mechanical verification of FSM descriptions. Using the ACL2 functional logic, we have defined a syntax well-formedness predicate and a symbolic simulator that defines the **DE2** cycle-based simulation semantics. **DE2** is deeply embedded within ACL2, and the **DE2** language includes an annotation facility that can be used by programs that manipulate **DE2** descriptions; this facility may restrict the use of defined modules or it may provide other module information. The **DE2** user may write and prove the correctness of programs that generate **DE2** descriptions. We have used **DE2** to mechanically verify components of the TRIPS microprocessor implementation.

1 Introduction

We present a formal description of and proof mechanism for the **DE2** hierarchical, occurrence-oriented finite state machine (FSM) description language, that we use to design and verify FSM-based designs or to optimize existing designs in a provably correct manner. This definition is primarily aimed at the representation and verification of hardware circuits, but could other areas such as protocols and software processes. Defining a hardware description language (HDL) is difficult because of the many different ways in which it may be used; for example, a HDL may be used to specify a simulation semantics, define what circuits can be specified, restrict allowable names, enforce circuit interconnect types, estimate power consumption, and provide layout or other manufacturing information. We have formally described the **DE2** language using the ACL2 logic [16], and we have formally verified **DE2** descriptions using the ACL2 theorem prover.

DE2 is designed to permit the rigorous hierarchical description and hierarchical verification of finite-state machines (FSMs). We call our language **DE2** (**D**ual-**E**val **2**) because of the two-pass approach that we employ for the language recognizers and evaluators. **DE2** is actually a general-purpose language for specifying FSMs; users may define their own language primitives. We recognize valid **DE2** descriptions with an ACL2 predicate that defines the permissible syntax, names, and hierarchy, of valid descriptions. The **DE2** language also provides a

rich annotation language that can be used to enforce syntactic and semantic design restrictions.

We begin our presentation by listing **DE2** language characteristics, contrasting the **DE2** language with other related efforts, and presenting some **DE2** language examples. We present the definition of its simulation-based semantics. We conclude by describing how we use the **DE2** language to verify circuits from the TRIPS microprocessor design [7].

2 DE2 Language Features

The development of **DE2** required balancing many demands. In particular, the demand for hardware verification requires that it be as simple as possible to evaluate, translate, and extend. In this section we list the resulting characteristics of **DE2**.

- **Hierarchical:** A module is defined by connecting submodules. Circuits may be defined in terms of modules that are small and easily verified.
- **Occurrence-Oriented:** Each reference to a previously defined module or primitive is called an occurrence. All defined modules are defined as a sequence of occurrences.
- **Deep Embedding in ACL2: DE2** modules are represented as ACL2 constants. Using the terminology defined by Boulton et al. [13], **DE2** is deeply embedded in the ACL2 language. This embedding allows us to write ACL2 functions which simulate, analyze, generate, and manipulate **DE2** modules.
- **Annotation Mechanisms:** We use annotations to describe elements of a circuit which are not defined by evaluation (e.g. layout information). In **DE2**, annotations are first class objects.
- **Parameterized Finite Types:** In **DE2**, every module input and output is a bit vector of parameterized length. When the lengths of all the inputs and outputs are known, we may appeal to BDD- and SAT-based techniques for verification.
- **Two-pass Evaluation:** A **DE2** module is evaluated by twice interpreting its list of occurrences. This two-pass evaluation necessitates a level-order for the combination functions.
- **Representation of Internal State:** This representation limits us to designing FSMs, but greatly simplifies the design and verification of these machines.
- **User-defined Primitive Modules:** We allow users to define primitive modules, rather than requiring that primitive modules be built into the language.
- **User-selectable Libraries:** Sets of primitives can become libraries. Libraries can be loaded into similar projects, which allows reuse of modules and verification efforts from past projects.
- **Verified DE2 Language Generators:** We can write ACL2 functions which to **DE2** modules. Since the semantics of **DE2** have been formalized in

ACL2, these generation functions can be shown to always generate correct **DE2** code.

- **Hierarchical Verification:** Our verification process involves verifying properties of submodules and then using these properties to verify larger modules built from these submodules. This hierarchical technique allows us to avoid reasoning about the internals of complex submodules.
- **Books for Verification Support:** We have defined a number of ACL2 "books" to assist the verification of **DE2** modules. When loaded into the theorem prover, these books use the ACL2 semantics of **DE2** to verify properties of **DE2** modules. We have used these books on a number of verification projects, some of which involve the verification of ACL2 functions that generate **DE2** circuits.

3 Related Work

The hardware verification community has taken two approaches [13] to defining the semantics of circuits: shallow and deep embedding. Shallow embedding defines the semantics of a circuit description by translating it into some formal language. Deep embedding uses a formal language to define the syntax and semantics of a HDL by embedding its definition and representation into the formal language being used.

The **DE2** language presented here has been defined by deeply embedding it inside the ACL2 language, a primitive recursive functional subset of Lisp [17]. By embedding **DE2** within ACL2, we are given access to a theorem proving environment which has successfully verified large-scale hardware systems [8, 9]. The formalization of the **DE2** language is similar in style to the embedding of the `DUAL-EVAL` HDL in NQTHM [11] and the `DE` language in ACL2 [10]. The `DE` language is different from `DUAL-EVAL` in that it permits user-defined primitives, re-usable libraries, annotations, and contains a different structuring of data for state-holding elements. The **DE2** language contains the new features of `DE`, but also has a parameterized type system, a more sophisticated system for applying non user-defined primitives (implemented as ACL2 functions), and a more automated verification system.

In other hardware verification efforts with ACL2, hardware descriptions were translated directly to ACL2 models in the style of shallow-embedding [8, 9]. These efforts do not permit the syntactic analysis of the circuits so represented; that is, it is not possible to treat the circuit descriptions as data so a program may be used to analyze its suitability.

Tom Melham used the HOL system [12] to deeply embed some elements of a hardware description language [12]. Boyer and Hunt attempted to deeply embed a subset of VHDL in the ACL2 logic, but this specification grew to more than 100 pages of formal mathematics, and its usefulness became suspect. Deeply embedding a HDL into another language brings great analytical power at the cost of having to manage all of the logical formalisms required—but these formalisms represent the real complexity that are inherit in such languages and

in their associated analysis and simulation systems. To make such an embedding useful, a serious effort needs has to be made to ensure an absolute economy of complexity, and there needs to libraries that ease the use of such an embedding.

A significant amount of work has focused on the use of functional programming languages to simply the writing of HDL-based descriptions. Mary Sheeran has developed the language Lava [1] and she has used it to design fast multipliers [2]. The strengths of Lava is its facilities to write programs that generate hardware—similar to the ACL2 programs we write to generate **DE2** descriptions —and its ability to embed layout information in the Lava language—similar to annotations in **DE2**. The Lava implementation does not include an associated reasoning system, but a user can appeal to SAT procedures to compare one Lava description against another description.

Our recent verification methodology, which combines a SAT-based decision procedure with theorem proving, was partially inspired by the work at Intel combining symbolic trajectory evaluation with theorem proving. This work makes use of the functional languages Lifted-FL [4] and, most recently, reFLect [3]. Some of the ways **DE2** differs from these languages include its simpler semantics (e.g. two pass evaluation), its simple syntax, its close correspondence to a subset of Verilog, and its embedding within a general-purpose theorem prover.

4 Example

The use of the **DE2** language is similar to the use of other hardware description languages. Circuits are specified in a hierarchical manner, and the syntactic form of the hierarchical circuit description also defines the hierarchical structure of a description's associated state. Here we give an example of a **DE2** circuit specification, and describe some of the restrictions imposed by the **DE2** language.

Our **DE2** language definition is a tremendous abstraction of this physical reality. The **DE2** language defines finite-state machines by permitting a user to define primitive elements. For this section, we assume the definition of Boolean connectives and state-holding elements have already been given. Issues such as clocking, wire delay, race conditions, power distribution, and heat, have been largely ignored.

Informally, the **DE2** language hierarchically defines Mealy machines: the outputs and next state of every module is a function of its inputs and internal state. By successively repeating the evaluation of an identified FSM, the **DE2** system can be used to emulate typical finite-state machine operation. **DE2** language definitions are written with a Lisp-style syntax using the Lisp syntax permitted for writing constant expressions; that is, modules definitions are represented as Lisp data, and they are not Lisp function definitions, macros, or other such constructs. We first give an example of several combinational circuits, where we exhibit some of the restrictions our evaluation approach imposes. Later we exhibit a sequential circuit.

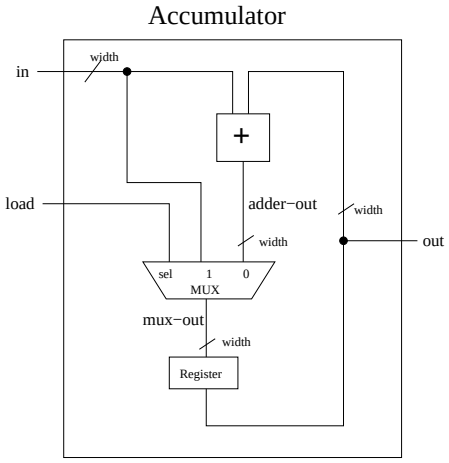

Fig. 1. Schematic of an Accumulator

4.1 Combinational Modules

Consider the circuit shown in Figure 1. In **DE2**, this circuit is represented as follows.

```
(accumulator
 (params width)
 (outs (out width))
 (ins (in width) (load 1))
 (wires (adder-out width) (mux-out width))
 (sts reg)
 (labels (out 'data) (in 'data) (adder-out 'data)
         (mux-out 'data) (load 'control))
 (occs
  (reg (out) (register width 'data) (mux-out))
  (adder (adder-out) (bufn width 'data) ((bv-adder width in out)))
  (mux (mux-out) (bufn width 'data) ((bv-if load in adder-out)))))
```

A module is identified by its name, in this case `accumulator`. Each module is composed of a set of key-value pairs whose entries depend on the type of the module. All modules have parameters, inputs, states, and outputs lists, identified by `params`, `ins`, `sts`, and `outs`, respectively. This module also has a `labels` entry, which is an annotation. Annotations are not required, but can be used to enable optimizations, assist verification, or provide information to other tools. In this case, we use the `labels` annotation, along with a static checker, to ensure that we do not use a `data` wire when a `control` wire was expected or vice versa. Annotations can also be used to represent layout information or other physical attributes – a user may define their own annotations.

A module will also include occurrences which define the relationship between its inputs, outputs, and internal modules. Each occurrence consists of a unique

occurrence name, a list of outputs, a module reference combined with its parameter list, and a list of inputs. For example, the first occurrence in the above example is named `reg`. The `reg` occurrence consists of an instance of a `register` module with the parameter `width`, input `mux-out`, and output `out`. The fact the `reg` occurs in the `accumulator` module's `sts` list denotes that it is a state-holding occurrence. Each input consists of an ACL2 expression of the inputs and internal "wires" of the module. Our primitive simulation-based evaluator only defines a finite list of ACL2 functions (e.g. bv-adder and bv-if) for use in such an expression.

The **DE2** language evaluation semantics define the outputs of a module as a function of its inputs and internal state. The next state of a module is also a function of a module's inputs and internal state. Evaluation is discrete; that is, there is an implicit notion of time which is broken into discrete steps.

Module evaluation begins by binding input values to a module's inputs, and binding state values to a module's states. Each occurrence is then evaluated in the order of its appearance. An occurrence is evaluated by binding its inputs and state to the specified arguments and then evaluating the reference itself. For the module defined above, the occurrence `reg` is evaluated first; the output of a register depends only on its internal state, not its inputs. After the value of `mux-out` is determined by evaluating the `mux` occurrence then internal state of the `reg` occurrence is updated.

In Section 6.1 we present some properties of this example which we have proven mechanically. Using the ACL2 theorem prover, we prove that for any data-path width a LOAD of A (i.e. `load` is high, `in` is A) followed by an ADD of B (i.e. `load` is low, `in` is B) produces the addition of A and B.

4.2 Primitives

A primitive module, corresponding to a hardware component built-in to a synthesis tool, has a definition in **DE2** that a non-primitive module. The difference between a primitive module is that rather than being defined in terms of occurrences of submodules, a primitive module is defined by lisp functions accessed through lambda modules. A lambda module has formals corresponding to the occurrence's list of parameters followed by the occurrence's list of inputs. The lambda module evaluates to a list with its first element being the state of the lambda module followed by its outputs. For example, the following is a definition of the primitive modules `bufn`, which is a submodule of our accumulator.

```
(bufn
 (type primitive)
 (params n sig-type)
 (outs (q n))
 (ins (x n))
 (labels (q sig-type) (x sig-type))
 (occs (st (q)
           ((lambda (x) (list 'nil x)))
           (x))))
```

The `bufn` module instantiates a single lambda module. Since the `bufn` module has no state, this lambda expression evaluates to a list whose first element is `nil`. The output of the `bufn` module, which corresponds to the second element of the list, is equal to its input. The other primitive found in our accumulator example, `register`, is defined as follows.

```
(register
 (type primitive)
 (params width sig-type)
 (outs (q width))
 (ins (d width))
 (sts st)
 (st-decls (st width))
 (labels (q sig-type) (d sig-type))
 (occs
  (st (q)
      ((lambda (width st d) (list d st)) width)
      (st d))))
```

The `register` example shows how a state-holding primitive is defined in **DE2**. The state of the `register` module is accessed through a lambda module named `st`, which turns the implicit input and output of state into an explicit input and output. The lambda module returns its input `d` as the next state and its state `st` as its output. Note that the `register` module also has a new field `st-decls`, which declares that the state element `st` is a bit-vector of length `width`. This declaration is not a requirement of **DE2** modules, but enables the later use of decision procedures.

5 The DE2 Evaluator

The definition of the **DE2** evaluator is composed of two groups, each containing two mutually recursive functions. These four functions implement the entire hierarchical evaluation of the outputs and next-state values for any well-formed hierarchical FSM defined using the **DE2** language, except for the evaluation of the lambda and ACL2 (primitive) expressions. This set of functions was designed with a number of different goals in mind, so design decisions were made to attempt to implement the desired properties while keeping the size of the system as small as possible.

The **DE2** language can be thought of as having two parts: primitive operations and interconnect. We have defined different primitive evaluators, depending on our needs. The primitive evaluator we use for verification of gate-like primitives interprets such primitive modules by applying ordinary Boolean operations. If we are interested in the fan-out of a set of signals, we use a different primitive evaluator. If we want to generate a count of the number of and type of primitive modules required to implement a referenced module, we use a primitive evaluator that collects that information from every primitive encountered during an evaluation pass – note that this does not just count the number of defined modules, but it counts the number of every kind of modules required to realize the

FSM being evaluated. If we want to compute a crude delay or power estimate, we use other primitive evaluators.

The semantic evaluation of a **DE2** design proceeds by binding actual (evaluated) parameters (both inputs and current states) to the formal parameters of the module to be evaluated; this in turn causes the evaluation of each submodule. This process is repeated recursively until a primitive module is encountered, and the specified primitive evaluator is called after binding the necessary arguments. This part of the evaluation can be thought of as performing all of the "wiring"; values are "routed" to appropriate modules and results are collected and passed along to other modules or become primary outputs. This set of definitions is composed of four (two groups of) functions (given below), and these functions contain an argument that permits different primitive evaluators to be used.

The following four functions completely define the evaluation of a netlist of modules, no matter which type of primitive evaluation is specified. The functions presented in this section constitute the entire definition of the simulator for the **DE2** language. This definition is small enough to allow us to reason with it mechanically, yet it is rich enough to permit the definition of a variety of evaluators. The `se` function evaluates a module and returns its primary outputs as a function of its inputs. The `de` function evaluates a module and returns its next state; this state will be structurally identical to the module's current state, but with updated values. Both `se` and `de` have sibling functions, `se-occ` and `de-occ` respectively, that iterate through each sub-module referenced in the body of a module definition. We present the `se`–`de` evaluator functions to make clear the importance we place on making the definition compact.

The `se` and `de` functions both have a `flg` argument that permits the selection of a specific primitive evaluator. The `fn` argument identifies the module name of a module to evaluate; its definition should be found in the `netlist`. The `ins` and `st` arguments provide the primary inputs and the current state of the module `fn` to be evaluated. The `params` argument allows for parameterized modules; that is, it is possible to define modules with wire and state sizes that are determined by this parameter. The `env` argument permits configuration or test information deep to be passed deep into the evaluation process.

The `se-occ` function evaluates each occurrence and returns an environment that includes values for all internal signals. The `se` function returns a list of outputs by filtering the desired outputs from this environment. To compute the outputs as functions of the inputs, only a single pass is required.

```
(defun se (flg fn params ins st env netlist)
  (if (consp fn)
      ;; Primitive Evaluation.
      (cdr (flg-eval-lambda-expr flg fn params ins env))
    ;; Evaluate submodules.
    (let ((module (assoc-eq fn netlist)))
      (if (atom module)
          nil
        (let-names
         (m-params m-ins m-outs m-sts m-occs)
```

```
        (m-body module)
        (let*
          ((new-env     (add-pairlist m-params params nil))
           (new-env     (add-pairlist (strip-cars m-ins)
                                      (flg-eval-list flg ins env)
                                      new-env))
           (new-env     (add-pairlist m-sts
                                      (flg-eval-expr flg st env)
                                      new-env))
           (new-netlist (delete-assoc-eq-netlist fn netlist)))
          (assoc-eq-list-vals
           (strip-cars m-outs)
           (se-occ flg m-occs new-env new-netlist))))))))

(defun se-occ (flg occs env netlist)
  (if (atom occs)  ;; Any more occurrences?
      env
    ;; Evaluate specific occurrence.
    (let-names
     (o-name o-outs o-call o-ins)
     (car occs)
     (se-occ flg (cdr occs)
             (add-pairlist
              (o-outs-names o-outs)
              (flg-eval-list
               flg (parse-output-list
                    o-outs
                    (se flg (o-call-fn o-call)
                        (flg-eval-list flg
                                       (o-call-params o-call)
                                       env)
                        o-ins o-name env netlist))
               env)
              env)
             netlist))))
```

Similarly, the functions `de` and `de-occ` perform the next-state computation for a module evaluation; given values for the primary inputs and a structured state argument, these two functions compute the next state of a specified module. This result state is structured isomorphically to its input's state. Note that the definition of `de` contains a reference to the function `se-occ`; this reference computes the value of all internal signals for the module whose next state is being computed. This call to `se-occ` represents the first of two passes through a module description when `DE` is computing the next state.

```
(defun de (flg fn params ins st env netlist)
  (if (consp fn)
      (car (flg-eval-lambda-expr flg fn params ins env))
    (let ((module (assoc-eq fn netlist)))
      (if (atom module)
```

```
          nil
        (let-names
         (m-params m-ins m-sts m-occs) (m-body module)
         (let*
           ((new-env     (add-pairlist m-params params nil))
            (new-env     (add-pairlist (strip-cars m-ins)
                                       (flg-eval-list flg ins env)
                                       new-env))
            (new-env     (add-pairlist m-sts
                                       (flg-eval-expr flg st env)
                                       new-env))
            (new-netlist (delete-assoc-eq-netlist fn netlist))
            (new-env     (se-occ flg m-occs new-env new-netlist)))
           (assoc-eq-list-vals
            m-sts
            (de-occ flg m-occs new-env new-netlist))))))))

(defun de-occ (flg occs env netlist)
  (if (atom occs)
      env
    (let-names
     (o-name o-call o-ins) (car occs)
     (de-occ flg (cdr occs)
             (cons
              (cons
               o-name
               (de flg (o-call-fn o-call)
                   (flg-eval-list flg (o-call-params o-call) env)
                   o-ins o-name env netlist))
              env)
             netlist))))
```

This completes the entire definition of the **DE2** evaluation semantics. This clique of functions is used for all different evaluators; the specific kind of evaluation is determined by the `flg` input. We have proved a number of lemmas that help to automate the analysis **DE2** modules. These lemmas allow us to hierarchically verify FSMs represented as **DE2** modules. We have also defined functions that repeatedly reference these functions so we can simulate a **DE2** design through any number of cycles.

An important aspect of this language semantics is its brevity; it is formal, and it provides a semantics for any FSM defined using the **DE2** language. Then, by using the ACL2 theorem prover, we can mechanically and hierarchically verify properties about any system defined using the **DE2** language.

6 Our Use of the DE2 System

Having an evaluator for **DE2** written in ACL2 enables many forms of verification. In Figure 2 we illustrate our verification system, which is built around the **DE2** language.

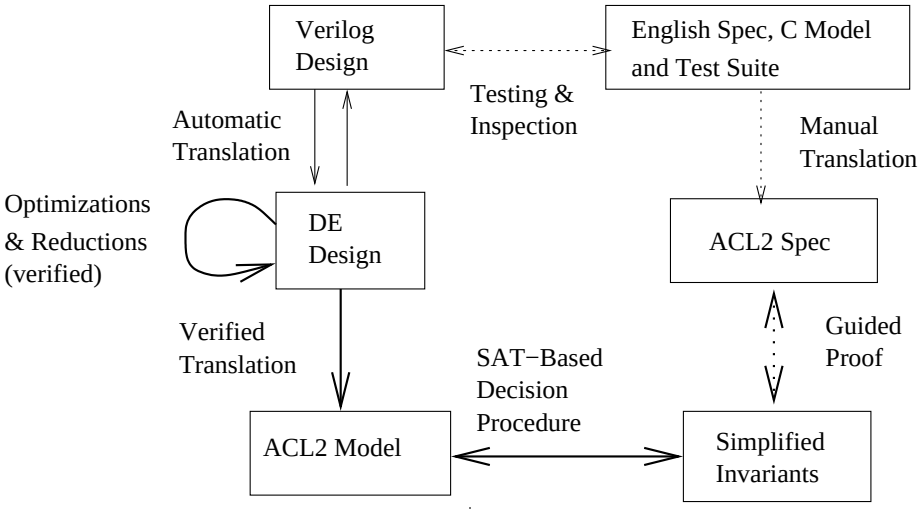

Fig. 2. An overview of the **DE2** verification system

We typically use the **DE2** verification system to verify Verilog designs. These designs are denoted in the upper left of Figure 2. Currently, the subset of Verilog includes arrays of wires (bit vectors), instantiations of modules, assignment statements, and a number of basic primitives (e.g. `&`, `?:` and `|`). We also allow the instantiation of memory (array) modules and vendor-defined primitives.

We have built a translator that translates a Verilog description into an equivalent **DE2** description. Our translator parses the Verilog source text into a Lisp expression, and then an ACL2 program converts this Lisp expression into a **DE2** description.

We have also built a translator that converts a **DE2** netlist into a cycle-accurate ACL2 model. This translator also an ACL2 proof that the **DE2** description is equivalent to the mechanical produced ACL2 model. The process of translating a **DE2** description into its corresponding ACL2 model may include cone-of-influence reductions; an ACL2 function is created for each module's output and irrelevant parts of the initial design are removed. This translator allows us to enjoy both the advantages of a shallow embedding (e.g. straightforward verification) and the advantages of a deep embedding (e.g. syntax resembling Verilog).

We start with an informal specification of the design in the form of English documents, charts, graphs, C-models, and test code which is represented in the upper right of Figure 2. This information is converted manually into a formal ACL2 specification. Using the ACL2 theorem prover, these specifications are simplified into a number of invariants and equivalence properties. If these properties are simple enough to be proven by our SAT-based decision procedure, we prove them automatically; otherwise, we simplify such conjectures using the

ACL2 theorem prover until we can again appeal to some automated decision procedure.

We also use our system to verify sets of **DE2** descriptions. This is accomplished by writing ACL2 functions that generate **DE2** descriptions, and then proving that these functions always produce circuits that satisfy their ACL2 specifications.

Since **DE2** descriptions are represented as ACL2 constants, functions that transform **DE2** descriptions can be verified using the ACL2 theorem prover. By converting from Verilog to **DE2** and from **DE2** to back into Verilog, we can use **DE2** as an intermediate language to perform verified optimizations. Another use of this feature involves performing reductions or optimizations on **DE2** specifications prior to verification. For example, one can use a decision procedure to determine that two **DE2** circuits are equivalent and then use this fact to avoid verifying properties of a less cleanly structured description.

We can also build static analysis tools, such as extended type checkers, in **DE2** by using annotations. In **DE2**, annotations are first-class objects (i.e. annotations are not embedded in comments). Therefore an annotation, such as the `labels` annotation in Section 4, is parsed as easily as any core language features. Such static checkers, since they are written in ACL2, can be analyzed and can also assist in the verification of **DE2** descriptions. Furthermore, annotations can be used to embed information into a **DE2** description to assist with synthesis.

6.1 Verification Example

To verify the **DE2** circuit in Section 4, we first generate an ACL2 model which is equivalent to the **DE2** circuit. The following theorems, which are proven automatically by a proof generated by our translator, prove that the ACL2 functions `accumulator-next-st` and `accumulator-out` produce the next state and the `out` output of the accumulator module.

```
(defthm accumulator-de-rewrite
  (implies (accumulator-& netlist)
           (equal (de flg 'accumulator
                      params in-exprs st-expr env netlist)
                  (let ((st (flg-eval-expr flg st-expr env))
                        (in (get-nth-value 0 flg in-exprs env))
                        (load (get-nth-value 1 flg in-exprs env))
                        (width (nth 0 params)))
                       (accumulator-next-st st width in load)))))

(defthm accumulator-se-rewrite
  (implies (accumulator-& netlist)
           (equal (se flg 'accumulator
                      params in-exprs st-expr env netlist)
                  (let ((st (flg-eval-expr flg st-expr env)))
                    (list (accumulator-out st))))))
```

We now can prove properties about the ACL2 model using the ACL2 theorem prover. For example, consider the following theorem:

```
(thm
 (let* ((state1 (accumulator-next-st state0 width A (LOAD)))
        (state2 (accumulator-next-st state1 width B (ADD))))
   (equal (accumulator-out state2) (bv-adder width a b))))
```

In this theorem, `state1` is the state of our accumulator after an arbitrary `LOAD` instruction (i.e. the `load` input to the accumulator is high), and `state2` is the state after following this `LOAD` with an `ADD` instruction (i.e. the `load` input is low). The theorem then states that the output of the accumulator is the addition of each cycles' inputs. We proved this theorem using the ACL2 theorem prover for any `width` accumulator. If we choose a specific width (e.g. a 32-bit accumulator), then this theorem can be proven automatically with our SAT-based decision procedure.

6.2 Verifying Components of the TRIPS Processor

We are using our verification system to verify components of the TRIPS processor. The TRIPS microprocessor is a prototype next-generation processor being designed by a joint effort between the University of Texas and IBM [7]. One novel aspect of the TRIPS microprocessor is that its memory is broken up into four pieces; each piece of memory has a separate cache and Load Store Queue (LSQ). We plan to verify the LSQ design, based on the design described in Sethumadhavan et al [6], using our verification system. We have already verified properties of its Data Status Network (DSN) component.

The DSN hardware provides the communication and buffering between four LSQ instances. Its design consists of 584 lines of Verilog code (including around 200 lines of comments), which we compile into a 427-line **DE2** description (with no comments). We use our verifying compiler to translate this **DE2** description into an ACL2 model and then prove the equivalence of the **DE2** description and its ACL2 specification. Using a mixture of theorem proving and a SAT-based decision procedure, we have proved properties that relate the output of the four DSN instances, communicating with each other over multiple cycles, to the output of a simplified machine; this simplified machine specifies the output that would be immediately produced if the communication were instantaneous.

7 Conclusion

The definition of the **DE2** language provides a user with a hierarchical language for specifying FSMs. By deeply embedding the definition of **DE2** within the ACL2 functional logic, we have provided a proof theory for verifying **DE2** module descriptions with respect to a number of primitive interpretations. The extensible structure of the **DE2** language and its general-purpose annotation

language allow a user to embed other types of information, such as a module's size, specification, layout, power requirements, and signal types. Instead of just verifying large netlists, we often compare netlists or transform one netlist into another netlist in a provable correct manner. We have extended the ACL2 theorem-proving system with a SAT procedure that can provide counter examples. We also have proved the correctness of functions that automatically generate circuits; this can greatly reduce the amount of **DE2** module definitions written by a user.

We believe that the design of **DE2** more closely fulfills the needs of modern hardware design and specification better than more traditional HDLs. The increasing demands placed on hardware or FSM specification languages is presently being served by embedding all kinds of extra information in the form of comments into a traditional HDL. This process forces non-standard, non-portable use of HDLs, and prevents there from being a single design description that can be accessed by all pre- and post-silicon development tools. We believe that **DE2** is the first formal attempt to integrate disparate design data into a single formalism. We believe future design systems should include similar features.

The **DE2** language, annotation system, and semantics provide a user with a uniform means of specifying and verifying a wide variety of both functional and extrinsic properties. We continue to expand the size and type of designs that we have verified. In the future, we want to use **DE2** to capture existing design elements to ease the reuse problem. Typically, in an industrial design flow, when a previously designed and verified design element is used in a new design, the verification has to be completely redone. Our ability to specify and verify modules in a hierarchical manner permits the reuse of prior verifications, and perhaps this verification reuse is the real key. Being able to reuse the design and the effort required to validate it will greatly reduce the effort of reusing previously designed modules.

References

1. Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware Design in Haskell. *The International Conference on Functional Programming (ICFP)*, pages 174–184, Volume 32, Number 1, ACM Press, 1998.
2. Mary Sheeran. Generating Fast Multipliers Using Clever Circuits. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, pages 6–20, LNCS, Volume 3312, Springer Verlag, 2004.
3. Sava Krstic and John Matthews. Semantics of the reFLect Language. *Principles and Practice of Declarative Programming (PPDP)*, pages 32–42, ACM Press, 2004.
4. Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving. *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS, Volume 1690, Springer Verlag, 1999.
5. Mark D. Aagaard, Robert B. Jones, John W. O'Leary, Carl-Johan H. Seger, and Thomas F Melham. A methodology for large-scale hardware verification. In Warren A. Hunt, Jr. and Steve Johnson, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, LNCS, Volume 1954, Springer Verlag, 2000.

6. S. Sethumadhavan, R.Desikan, D.Burger, C.R.Moore and S.W.Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors (Load/Store Queue Design). *36th International Symposium on Microarchitecture (MICRO 36)*, pages 399–410, 2003.
7. The Tera-op Reliable Intelligently adaptive Processing System(TRIPS), http://www.cs.utexas.edu/users/cart/trips/
8. Bishop Brock, Matt Kaufmann, and J Moore. ACL2 Theorems about Commercial Microprocessors. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293, LNCS, Volume 1166, Springer-Verlag, 1996.
9. Jun Sawada. Formal Verification of an Advanced Pipelined Machine. PhD Thesis, University of Texas at Austin, 1999.
10. Warren A. Hunt, Jr. The DE Language. *Computer-aided Reasoning: ACL2 case studies*, pages 151–166, Kluwer Academic Publishers, 2000.
11. Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, Boston, 1988.
12. M. J. C. Gordon and T. F. Melham (editors). *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.
13. Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with Embedding Hardware Description Languages in HOL, *Theorem Provers in Circuit Design*, pages 129–156, IFIP Transactions A-10, Elsevier Science Publishers, 1992.
14. Mike Gordon. Why Higher-order Logic is a Good Formalism for Specifying and Verifying Hardware. Technical Report 77, University of Cambridge, Computer Laboratory, 1985.
15. Warren A. Hunt, Jr. and Bishop C. Brock. A Formal HDL and Its Use in the FM9001 Verification. In C.A.R. Hoare and M.J.C. Gordon, editors, Mechanized Reasoning and Hardware Design, pages 35–48, Prentice-Hall International Series in Computer Science, 1992.
16. Matt Kaufmann and J Strother Moore. ACL2: An Industrial Strength Version of NQTHM. *Eleventh Annual Conference on Computer Assurance (COMPASS-96)*, pages 23–34, IEEE Computer Society Press, 1996.
17. Guy Steele. Common Lisp: The Lanugage, Second Edition. Digital Press, 1990.
18. Phillip J. Windley and Michael L. Coe. A Correctness Model for Pipelined Microprocessors, *Theorem Provers in Circuit Design : Theory, Practice, and Experience*, LNCS, Volume 901, Springer Verlag, pages 33-51, 1995.