Modeling Asynchronous Circuits in ACL2 Using the Link-Joint Interface

Cuong Chau

ckcuong@cs.utexas.edu

Department of Computer Science

The University of Texas at Austin

April 19, 2016

Outline

1 Introduction

2 The DE Language

3 The Link-Joint Interface

4 Future Work

Outline

1 Introduction

2 The DE Language

3 The Link-Joint Interface

4 Future Work

Introduction

Synchronous circuits (Clocked circuits): changes in the state of storage elements are synchronized by **a global clock signal**.

Asynchronous circuits (Self-timed circuits): there is no global clock signal distributed in asynchronous circuits. The communication between components is performed via **local handshake protocols**.

Introduction

Synchronous circuits (Clocked circuits): changes in the state of storage elements are synchronized by **a global clock signal**.

Asynchronous circuits (Self-timed circuits): there is no global clock signal distributed in asynchronous circuits. The communication between components is performed via **local handshake protocols**.

Why asynchronous?

Introduction

Synchronous circuits (Clocked circuits): changes in the state of storage elements are synchronized by **a global clock signal**.

Asynchronous circuits (Self-timed circuits): there is no global clock signal distributed in asynchronous circuits. The communication between components is performed via **local handshake protocols**.

Why asynchronous?

- Low power consumption.
- High operating speed.
- Better composability and modularity.
- ...

Introduction

Formal verification in asynchronous systems is an active research area.

Our goal: developing a comprehensive verification strategy for verifying asynchronous systems in ACL2.

- Leveraging existing work in the clocked design paradigm.

- Developing new approaches for modeling asynchronous systems in ACL2. Dealing with:

Introduction

Formal verification in asynchronous systems is an active research area.

Our goal: developing a comprehensive verification strategy for verifying asynchronous systems in ACL2.

- Leveraging existing work in the clocked design paradigm.
    - The DE language - a hardware description language written in ACL2 [W. Hunt, 2000].
- Developing new approaches for modeling asynchronous systems in ACL2. Dealing with:

Introduction

Formal verification in asynchronous systems is an active research area.

Our goal: developing a comprehensive verification strategy for verifying asynchronous systems in ACL2.

- Leveraging existing work in the clocked design paradigm.
  - The DE language - a hardware description language written in ACL2 [W. Hunt, 2000].
- Developing new approaches for modeling asynchronous systems in ACL2. Dealing with:
  - no global clock signal,

Introduction

Formal verification in asynchronous systems is an active research area.

Our goal: developing a comprehensive verification strategy for verifying asynchronous systems in ACL2.

- Leveraging existing work in the clocked design paradigm.
    - The DE language - a hardware description language written in ACL2 [W. Hunt, 2000].
- Developing new approaches for modeling asynchronous systems in ACL2. Dealing with:
    - no global clock signal,
    - local handshake protocols,

Introduction

Formal verification in asynchronous systems is an active research area.

Our goal: developing a comprehensive verification strategy for verifying asynchronous systems in ACL2.

- Leveraging existing work in the clocked design paradigm.
    - The DE language - a hardware description language written in ACL2 [W. Hunt, 2000].
- Developing new approaches for modeling asynchronous systems in ACL2. Dealing with:
    - no global clock signal,
    - local handshake protocols,
    - non-deterministic behavior due to *uncertain but bounded delays* on wires and gates,

Introduction

Formal verification in asynchronous systems is an active research area.

Our goal: developing a comprehensive verification strategy for verifying asynchronous systems in ACL2.

- Leveraging existing work in the clocked design paradigm.
    - The DE language - a hardware description language written in ACL2 [W. Hunt, 2000].
- Developing new approaches for modeling asynchronous systems in ACL2. Dealing with:
    - no global clock signal,
    - local handshake protocols,
    - non-deterministic behavior due to *uncertain but bounded delays* on wires and gates,
    - ...

Introduction

Formal verification in asynchronous systems is an active research area.

Our goal: developing a comprehensive verification strategy for verifying asynchronous systems in ACL2.

- Leveraging existing work in the clocked design paradigm.
  - The DE language - a hardware description language written in ACL2 [W. Hunt, 2000].
- Developing new approaches for modeling asynchronous systems in ACL2. Dealing with:
  - no global clock signal,
  - local handshake protocols,
  - non-deterministic behavior due to *uncertain but bounded delays* on wires and gates,
  - ...

Approach

Developing new approaches for modeling asynchronous systems in ACL2. Dealing with:

- no global clock signal,
- local handshake protocols,

Approach

Developing new approaches for modeling asynchronous systems in ACL2. Dealing with:

- no global clock signal,
  $\Rightarrow$ Every state-holding device in the DE primitive database must be governed by its own clock signal.
- local handshake protocols,

Approach

Developing new approaches for modeling asynchronous systems in ACL2. Dealing with:

- no global clock signal,
  $\Rightarrow$ Every state-holding device in the DE primitive database must be governed by its own clock signal.
- local handshake protocols,
  $\Rightarrow$ Modeling the link-joint interface introduced by Roncken et al. [M. Roncken et al., 2015] in DE.

Outline

1 Introduction

2 The DE Language

3 The Link-Joint Interface

4 Future Work

The DE Language

DE is a formal occurrence-oriented description language that permits the hierarchical definition of finite-state machines in the style of a hardware description language [W. Hunt, 2000].

The DE Language

DE is a formal occurrence-oriented description language that permits the hierarchical definition of finite-state machines in the style of a hardware description language [W. Hunt, 2000].

The semantics of the DE language is given by a simulator that, given the current inputs and current state for a module, will compute the module's current outputs and the next state.

The DE Language

DE is a formal occurrence-oriented description language that permits the hierarchical definition of finite-state machines in the style of a hardware description language [W. Hunt, 2000].

The semantics of the DE language is given by a simulator that, given the current inputs and current state for a module, will compute the module's current outputs and the next state.

A DE description is an ACL2 constant containing an ordered list of modules, which we call a netlist.

The DE Language

DE is a formal occurrence-oriented description language that permits the hierarchical definition of finite-state machines in the style of a hardware description language [W. Hunt, 2000].

The semantics of the DE language is given by a simulator that, given the current inputs and current state for a module, will compute the module's current outputs and the next state.

A DE description is an ACL2 constant containing an ordered list of modules, which we call a netlist.

Each module consists of five elements: a netlist-unique module name, inputs, outputs, internal states, and occurrences.

The DE Language

DE is a formal occurrence-oriented description language that permits the hierarchical definition of finite-state machines in the style of a hardware description language [W. Hunt, 2000].

The semantics of the DE language is given by a simulator that, given the current inputs and current state for a module, will compute the module's current outputs and the next state.

A DE description is an ACL2 constant containing an ordered list of modules, which we call a netlist.

Each module consists of five elements: a netlist-unique module name, inputs, outputs, internal states, and occurrences.

Each occurrence consists of four elements: a module-unique occurrence name, outputs, a reference to a primitive or defined module, and inputs.

DE Language Example

```
(defconst *half-adder*
  '((half-adder    ;; module name
     (a b)         ;; module inputs
     (sum carry)   ;; module outputs
     ()            ;; internal states
      ;; occurrences
     ((g0       ;; occurrence name
       (sum)    ;; occurrence outputs
       f-xor    ;; a primitive reference
       (a b))   ;; occurrence inputs
      (g1 (carry) f-and (a b))))))
```

DE Language Example

```
(defconst *full-adder*
  (cons
   '(full-adder
     (a b c)
     (sum carry)
     ()
     ((t0 (sum1 carry1) half-adder           (a b))
      (t1 (sum  carry2) half-adder        (sum1 c))
      (t2 (carry)       f-or       (carry1 carry2))))

   *half-adder*))
```

DE Language Example

```
(defconst *one-bit-counter*
  (cons
   '(one-bit-counter
     (clk carry-in reset-)
     (out carry)
     (g0)
     ((g0 (out)         ff          (clk sum-reset-))
      (g1 (sum carry)  half-adder   (carry-in out))
      (g2 (sum-reset-) f-and          (sum reset-))))

   *half-adder*))
```

The DE Simulator

The operational semantics for the DE language is given by the DE simulator, which is composed of two sets of mutually recursive functions.

The DE Simulator

The operational semantics for the DE language is given by the DE simulator, which is composed of two sets of mutually recursive functions.

- The mutually recursive functions `se` and `se-occ` compute the **outputs** of a module being evaluated given its inputs and its current states.

The DE Simulator

The operational semantics for the DE language is given by the DE simulator, which is composed of two sets of mutually recursive functions.

- The mutually recursive functions `se` and `se-occ` compute the **outputs** of a module being evaluated given its inputs and its current states.
- The mutually recursive functions `de` and `de-occ` compute the **next state** of a module being evaluated given its inputs and its current states.

The DE Simulator

The operational semantics for the DE language is given by the DE simulator, which is composed of two sets of mutually recursive functions.

- The mutually recursive functions `se` and `se-occ` compute the **outputs** of a module being evaluated given its inputs and its current states.
- The mutually recursive functions `de` and `de-occ` compute the **next state** of a module being evaluated given its inputs and its current states.

Demo.

An Example DE Proof

Prove the correctness of a parameterized ripple-carry adder.

```
(defun ripple-adder (c a b)
  (declare (xargs :guard (and (booleanp c)
                              (boolean-listp a)
                              (boolean-listp b))))
   ;; c is a bit, a and b are bit-vectors of some length n;
   ;; this function returns a bit vector of length n+1.
  (if (endp a)
      (list c)
    (cons (xor c (xor (car a) (car b)))
          (ripple-adder (or (and (car a) (car b))
                            (and (car a) c)
                            (and (car b) c))
                        (cdr a)
                        (cdr b)))))
```

An Example DE Proof

Prove the correctness of a parameterized ripple-carry adder.

```
(defun ripple-adder (c a b)
  (declare (xargs :guard (and (booleanp c)
                              (boolean-listp a)
                              (boolean-listp b))))
   ;; c is a bit, a and b are bit-vectors of some length n;
   ;; this function returns a bit vector of length n+1.
  (if (endp a)
      (list c)
    (cons (xor c (xor (car a) (car b)))
          (ripple-adder (or (and (car a) (car b))
                            (and (car a) c)
                            (and (car b) c))
                        (cdr a)
                        (cdr b)))))
```

Demo.

Outline

1 Introduction

2 The DE Language

3 The Link-Joint Interface

4 Future Work

Link and Joint

The dataflow in self-timed systems can be viewed as a directed graph with links as edges and joints as nodes [M. Roncken et al., 2015], where:

- Links are communication channels, with data flowing in the direction of the edges representing the links.
- Joints are modules that implement flow control and data operations.

Link and Joint

The dataflow in self-timed systems can be viewed as a directed graph with links as edges and joints as nodes [M. Roncken et al., 2015], where:

- Links are communication channels, with data flowing in the direction of the edges representing the links.
- Joints are modules that implement flow control and data operations.

Data are stored in links, not in joints. Data flow from one end of the link to the other end, and are captured in-between.

Link and Joint

A **link** receives `fill` or `drain` commands from and reports its full/empty state to its corresponding joints. When a link receives a `fill` command, it changes its state to full. A link will change to the empty state if it receives a `drain` command.

Link and Joint

A **link** receives `fill` or `drain` commands from and reports its full/empty state to its corresponding joints. When a link receives a `fill` command, it changes its state to full. A link will change to the empty state if it receives a `drain` command.

A **joint** receives the full/empty states of its links and issues the `fill` and `drain` commands when the handshake condition is satisfied. In particular, whenever its incoming links are full and its outgoing links are empty, it will perform the following three actions in parallel:

Link and Joint

A **link** receives `fill` or `drain` commands from and reports its full/empty state to its corresponding joints. When a link receives a `fill` command, it changes its state to full. A link will change to the empty state if it receives a `drain` command.

A **joint** receives the full/empty states of its links and issues the `fill` and `drain` commands when the handshake condition is satisfied. In particular, whenever its incoming links are full and its outgoing links are empty, it will perform the following three actions in parallel:

- hand-over data computed from the incoming links to the corresponding outgoing links,
- make the incoming links empty,
- make the outgoing links full.

Handshake Protocol Using the Link-Joint Interface

There is a feedback loop from link-state to joint-action and back to link-state.

Handshake Protocol Using the Link-Joint Interface

There is a feedback loop from link-state to joint-action and back to link-state.

Handshake protocols can be established in terms of the link-joint interface.

Handshake Protocol Using the Link-Joint Interface

There is a feedback loop from link-state to joint-action and back to link-state.

Handshake protocols can be established in terms of the link-joint interface.

Demo: Modeling a simple while-loop circuit in an asynchronous manner using the link-joint interface.

Outline

1 Introduction

2 The DE Language

3 The Link-Joint Interface

4 Future Work

Future Work

Develop more efficient decomposition-proof methods in asynchronous circuits.

Extend the DE system to modeling non-deterministic behavior in asynchronous circuits.

Implement tools and techniques for building and analyzing asynchronous circuits in a more automated manner.

References

W. Hunt (2000)
The DE Language
Computer-Aided Reasoning: ACL2 Case Studies, Kluwer Academic Publishers Norwell, MA, USA, 151 – 166.

M. Roncken, S. Gilla, H. Park, N. Jamadagni, C. Cowan, I. Sutherland (2015)
Naturalized Communication and Testing
ASYNC 2015, 77 – 84.

Thank You!