Fifth International Workshop on Designing Correct Circuits

Barcelona, 27–28 March 2004

A Satellite Event of the ETAPS 2004 group of conferences

Participants' Proceedings

Edited by Tom Melham and Mary Sheeran

Preface

This volume contains material provided by the speakers to accompany their presentations at the Fifth International Workshop on Designing Correct Circuits, held on 27th and 28th March 2004 in Barcelona. The workshop is a satellite event of the ETAPS group of conferences. Previous workshops in the informal DCC series were held in Oxford (1990), Lyngby (1992), Båstad (1996), and Grenoble (2002). Each of these stimulating meetings made a memorable contribution to building our research community.

The 2004 DCC workshop again brings together academic and industrial researchers in formal methods for hardware design and verification. It will allow participants to learn about the current state of the art in formally based hardware verification and it is intended to further the debate about how more effective design and verification methods can be developed.

Nowadays, much research in hardware verification takes place in industry as well as in academia. To make progress on the longer-term problems in our field, while keeping our work grounded in practical engineering problems, academic and industrial researchers must continue to work together on the problems facing microprocessor and ASIC designers, now and into the future. A major aim of the DCC series of workshops has been to provide a congenial and relaxed venue for communication with fellow researchers in our community. The programme that DCC 2004 has attracted will keep the debate stimulating and productive, and we look forward to two great days of presentations and discussion.

We would like to express our gratitude to the members of the programme committee for their work in selecting the presentations, and to all the speakers and participants for their contributions to designing correct circuits.

Tom Melham and Mary Sheeran
March 2004

Programme Committee

Koen Claessen (Chalmers)
Nicolas Halbwachs (IMAG)
Scott Hazelhurst (Witwatersrand)
Axel Jantsch (KTH)
Steve Johnson (Indiana)
Andy Martin (IBM)
John Matthews (OGI)
Tom Melham (Oxford)
John O'Leary (Intel)
Ellen Sentovich (Cadence)
Mary Sheeran (Chalmers)
Satnam Singh (Xilinx)
Jean Vuillemin (ENS, Paris)

DCC 2004 Programme

Saturday, 27 March 2004

9:00–9:40
A Hierarchical Modeling System
Warren A. Hunt, Jr. and Erik Reeber (University of Texas at Austin)
9:40–10:20
Wired—a Language for Describing Non-Functional Properties of Digital Circuits
Emil Axelsson, Koen Claessen, and Mary Sheeran (Chalmers University of Technology)

10:20–11:00 Coffee

11:00–10:40
Integrating Formal Methods with Digital Circuit Design in Hydra
John O'Donnell (University of Glasgow)
11:40–12:20
HML: A language for high-level design of high-frequency circuits
Andrew K. Martin (IBM)

12:20–14:00 Lunch

14:00–14:40
Late Design Changes (ECOs) for sequentially optimized high-level Esterel designs
Laurent Arditi, Gérard Berry, and Michael Kishinevsky (Esterel Technologies and Intel)
14:40–15:20
Structure-Driven Equivalence Verification for Circuits Optimized by Retiming and Combinational Synthesis
Maher Mneimneh and Karem Sakallah (University of Michigan)

15:20–16:00 Coffee

16:00–16:40
A Reflective Functional Language for Hardware Design and Theorem Proving
Jim Grundy, Tom Melham, and John O'Leary (Intel and University of Oxford)
16:40–17:20
Verifying the ARM Block Data Transfer Instructions
Anthony Fox (University of Cambridge)

Sunday, 28 March 2004

9:00–9:40
satGSTE: Combining the Abstraction of GSTE with the Capacity of a SAT Solver
Jin Yang, Rami Gil, and Eli Singerman (Intel)
9:40–10:20
Symbolic Trajectory Evaluation using Satisfiability Solvers
Koen Claessen and Jan-Willem Roorda (Chalmers University of Technology)

10:20–11:00 Coffee

11:00–11:40
Verification of Parametric Timed Circuits using Octahedra
Robert Clarisó and Jordi Cortadella (Universitaat Politécnica de Catalunya)
11:40–12:20
Trading Completeness for Capacity using Probabilistic Techniques
René Krenz and Elena Dubrova (Royal Institute of Technology, Stockholm)

12:20–14:00 Lunch

14:00–14:40
Formal Verification of Floating Point Multiply Add on Itanium Processor
Anna Slobodová and Krishna Nagalla (Intel)
14:40–15:20
The Post-Silicon Verification Problem: Designing Limited Observability Checkers for Shared Memory Processors
Ganesh Gopalakrishnan and Ching-Tsun Chou (University of Utah and Intel)

15:20–16:00 Coffee

16:00–16:40
An Operational Semantics for Safety PSL
Koen Claessen and Johan Mårtensson (Safelogic AB, Chalmers University of Technology, and Gothenburg University)
16:40–17:20
PSL semantics in higher order logic
Mike Gordon (University of Cambridge)

A Hierarchical Modeling System

*** Work in Progress ***

Warren A. Hunt, Jr. and Erik Reeber

CS and ECE Departments
1 University Station, M/S C0500
The University of Texas
Austin, TX 78712-0233

E-mail: {hunt,reeber}@cs.utexas.edu
TEL: +1 512 471 9748, +1 512 471 9749
FAX: +1 512 471 8885

A Hierarchical Modeling System

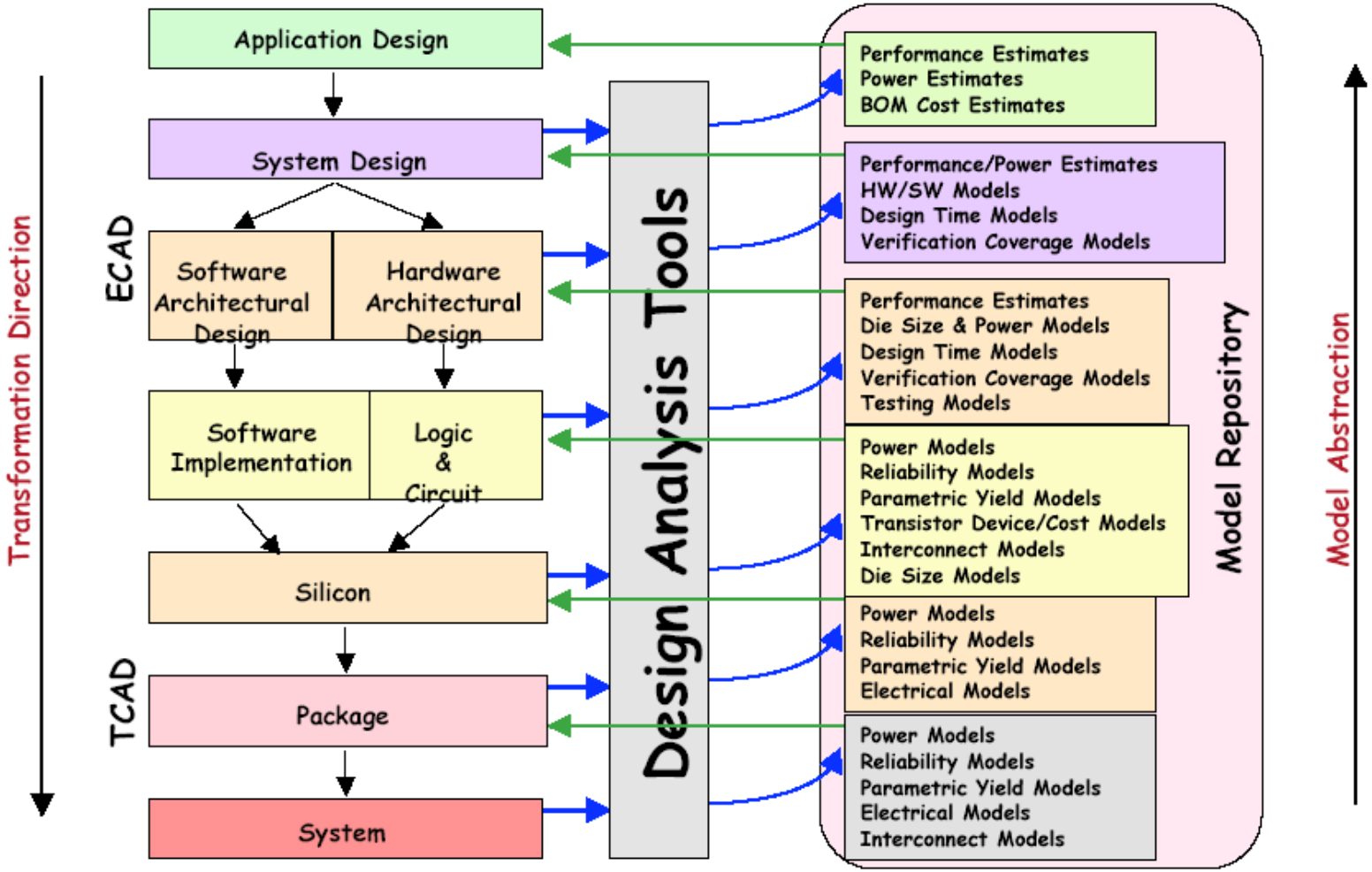
***** Work in Progress *****

We are developing a new hierarchical, occurrence-oriented hardware and software modeling language, `DE`, whose design includes mechanisms for formal functional, property, timing, and power specifications.

- Language is designed in the `DUAL-EVAL` style.
  - Hierarchical specification system for cooperating FSMs.
  - Possible to embed multiple libraries.
  - Annotation language available for specifying functional properties, as well as "down stream" tools hints.
  - Predicate recognizes well-formed DE system descriptions.
  - Formal simulator and "lambda" evaluator define the semenatics.

Purpose of this Effort

We want to address the specification and verification of several elements of the following diagram.



DE Language Characteristics

- We are trying to address problems that may arise in a "next generation" RTL.
  - Reuse of previously defined IP.
  - Heavy use of parameterized IP.
  - Rapid prototyping.
  - Non-functional property specification and verification.
  - First-class system annotations.
- Annotations
  - "Hook up" annotations, e.g., "active low" signaling.
  - Functional annotations, e.g., "one-hot" output vector.
  - Annotations can be manipulated.
- Parameterized.
  - Multiple parameters permited.
  - Evaluation semantics and environment.
  - Parameters can be passed to submodules.
- Interpreted functional semantics.
  - No compilation required, immediate execution.
  - Symbolic simulation capability.
  - Predicate recognized well formed systems.
  - Inter-changeable library primitives.

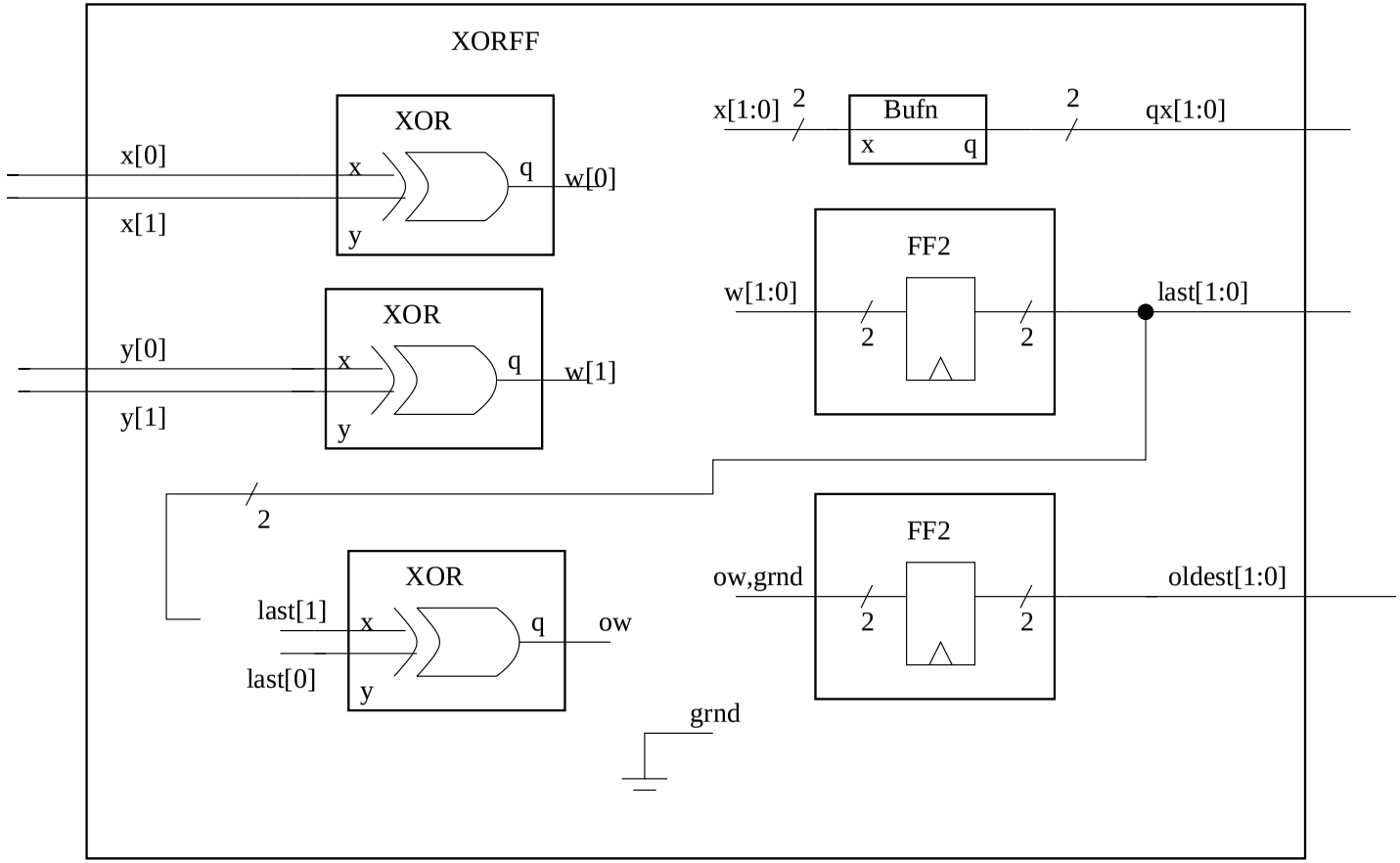
DE Language Example

```
(defun xor-* ()
  '(xor
    (type primitive)
    (params )
    (outs (q 1))
    (ins (x 1) (y 1))
    (sts )
    (wires )
    (verilog-code "   assign q = x^y;")
    (c-code
     "   if (testBit(&wires->X, 0)"
     "       ^ testBit(&wires->Y))"
     "       setOn(&wires->Q, 0);"
     "   else "
     "       setOff(&wires->Z, 0);")
    (occs
     (st ((q 0 0))
         ((lambda (x y)
            (list2
             'nil
             (or-list2
              (and-list2
               (not-list x) y)
              (and-list2 x (not-list y))))))
         ((g x 0 0) (g y 0 0))))))
```

- Primitive module
  - Lambda evaluator semantics
  - The definition of DE does not include any built-in primitives.
- Different target languages.
  - C-language simulation target.
    - Hierarchical C models.
    - Mechanical conversion of test vectors.
  - For UT TRIPS project, we are designing in DE and mechanically producing Verilog.
- Design of DE heavily influenced by need to interface with multiple target environments.

Hierarchical DE Language Example

The DE language has been designed to permit hierarchical verification.



Primitive DE Language Examples

- N-bit modules are possible.

```
(defun bufn-* ()
  '(bufn
    (type primitive)
    (params n)
    (outs (q n))
    (ins (x n))
    (sts )
    (wires )
    (verilog-code "   assign Q = X;" )
    (C-CODE "   wires->Q = wires->X;")
    (occs
     (st ((q 0 (1- n)))
         ((lambda (n x)
            (list2 'nil x)) n)
         ((g x 0 (1- n)))))))
```

- Every "wire-name" represents a one-dimentional vector.

```
(defun ff2-* ()
  '(ff2
    (type primitive)
    (params )
    (outs (q 2))
    (ins (x 2))
    (sts (st 2))
    (wires )
    (cc "Signal `clk' added.")
    (verilog-code
     "   trips_dff #(2) mem_block"
     "      (.q(q),"
     "       .clk(clk),"
     "       .d(x));")
    (C-CODE
     "   wires->Z = wires->ST;"
     "   if (change_state) "
     "      wires->ST = wires->IN;")
    (occs
     (st ((q 0 1))
         ((lambda (s x)
            (list2 x s)))
         ((g st 0 1) (g x 0 1))))))
```

Hierarchical DE Language Example

We now finally give a non-primitive module definition.

```
(defun xorff-* ()
  '(xorff
    (type module)
    (params )
    (outs  (qx 2) (last 2) (older 2))
    (ins   (x 2)  (y 2))
    (sts   (st-last 2) (st-older 2))
    (wires (w 2) (ow 1))
    (occs
     (st-last  ((last 0 1))  (ff2)    ((g w 0 1)))
     (xor-w0   ((w 0 0))     (xor)    ((g x 0 0) (g x 1 1)))
     (xor-w1   ((w 1 1))     (xor)    ((g y 0 0) (g y 1 1)))
     (st-older ((older 0 1)) (ff2)    ((a (g ow 0 0) '(nil))))
     (xor-ow   ((ow 0 0))    (xor)    ((g last 0 0) (g last 1 1)))
     (x-out    ((qx 0 1))    (bufn 2) ((g x 0 1))))))
```

DE Language Theorem

The following is representative of theorems we prove using DE.

- Our assumptions are:
  - Recognizer for identifying a suitable netlist.
  - Inputs are not equal.
  - We have sufficient clock.
- Our conclusion is the outputs are always '(nil nil).

```
(defthm nil-nil-after-three
  (implies (and (xorff-& netlist)
                (ev-xor-t-input-list flg n input-lst env)
                (not (zp n))
                (not (zp (1- n))))
           (equal (get-sublist
                   (caddr (car (run n flg 'xorff params
                                    input-lst st env netlist)))
                   0 1)
                  '(nil nil))))
```

The Definition of DE, Actually SE

```
(defun se (flg fn params ins sts env netlist)
  (if (consp fn)
      (cdr (flg-eval-expr flg (append (cons fn params) ins) env))
    (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
                               (flg-eval-list flg params env) nil))
                (new-env
                 (add-pairlist (strip-cars m-ins)
                               (flg-eval-list flg ins env)
                               new-env))
                (new-env
                 (add-pairlist (strip-cars m-sts)
                               (flg-eval-list
                                flg (parse-state-list m-sts sts
                                                      (list 'quote 0)) 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)))))))))
```

The Definition of DE, Actually SE, continued

This of SE is the part that evaluates each occurrence. The definition of DE is isomorphic and almost identical except that DE calls SE to compute the "wire values."

```
(defun se-occ (flg occs env netlist)
  (cond
   ((atom occs)
    env)
   ((equal (caar occs) 'cc)  ;; this occurrence is a comment
    (se-occ flg (cdr occs) env netlist))
   (t
    (let-names
     (o-name o-outs o-call o-ins)
     (car occs)
     (se-occ flg (cdr occs)
             (add-pairlist
              (strip-cars o-outs)
              (flg-eval-list flg (parse-output-list
                                  o-outs
                                  (se flg (o-call-fn o-call)
                                      (o-call-params o-call)
                                      o-ins o-name env netlist))
                             env)
              env)
             netlist)))))
```

The Definition of DE

```
(defun de (flg fn params ins sts env netlist)
  (if (consp fn)
      (car (flg-eval-expr flg (append (cons 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
                               (flg-eval-list flg params env) nil))
                (new-env
                 (add-pairlist (strip-cars m-ins)
                               (flg-eval-list flg ins env)
                               new-env))
                (new-env
                 (add-pairlist (strip-cars m-sts)
                               (flg-eval-list
                                flg (parse-state-list m-sts sts
                                                      (list 'quote 0)) env)
                               new-env))
                (new-netlist (delete-assoc-eq-netlist fn netlist))
                (new-env (se-occ flg m-occs new-env new-netlist)))
           (append-all
            (assoc-eq-list-vals (strip-cars m-sts)
                                (de-occ flg m-occs new-env new-netlist)))))))))
```

The Definition of DE, continued

```
(defun de-occ (flg occs env netlist)
  (cond
   ((atom occs) env)
   ((equal (caar occs) 'cc) ;; this occurance is a comment
    (de-occ flg (cdr occs) env netlist))
   (t
    (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) (o-call-params o-call)
                   o-ins o-name env netlist))
            env)
      netlist)))))

(defun run (n flg fn params input-list init-state env netlist)
  (if (zp n)
      (run-one-test flg fn params (car input-list) init-state env netlist)
    (run-one-test flg fn params (car input-list)
                  (list 'quote
                        (cadr (run (1- n) flg fn params (cdr input-list) init-state
                                   env netlist)))
                  env netlist)))
```

Verification of Module Generators

We use module generators to greatly reduce the amount of DE code that we need to write.

- Module generators can produce hierarchical designs.
  - We write ACL2 programs that generate modules.
  - We verify module generators using ACL2.

The DE approach provides us the opportunity to verify ACL2 programs that operate on DE modules.

- ACL2 programs that transform DE to DE can optimize DE code or simplify modules to ease verification.
- ACL2 programs that transform DE into conjunctive normal form (CNF) can generate input for a SAT solver.

SAT-based Equivalence-Checking

We have implemented an equivalence checker to compare `DE` modules.

- We convert `DE` modules into CNF form and pass the formulas to zChaff.
- If successful, we create an ACL2 rule that rewrites one module into another, under the condition that the inputs are well-formed.

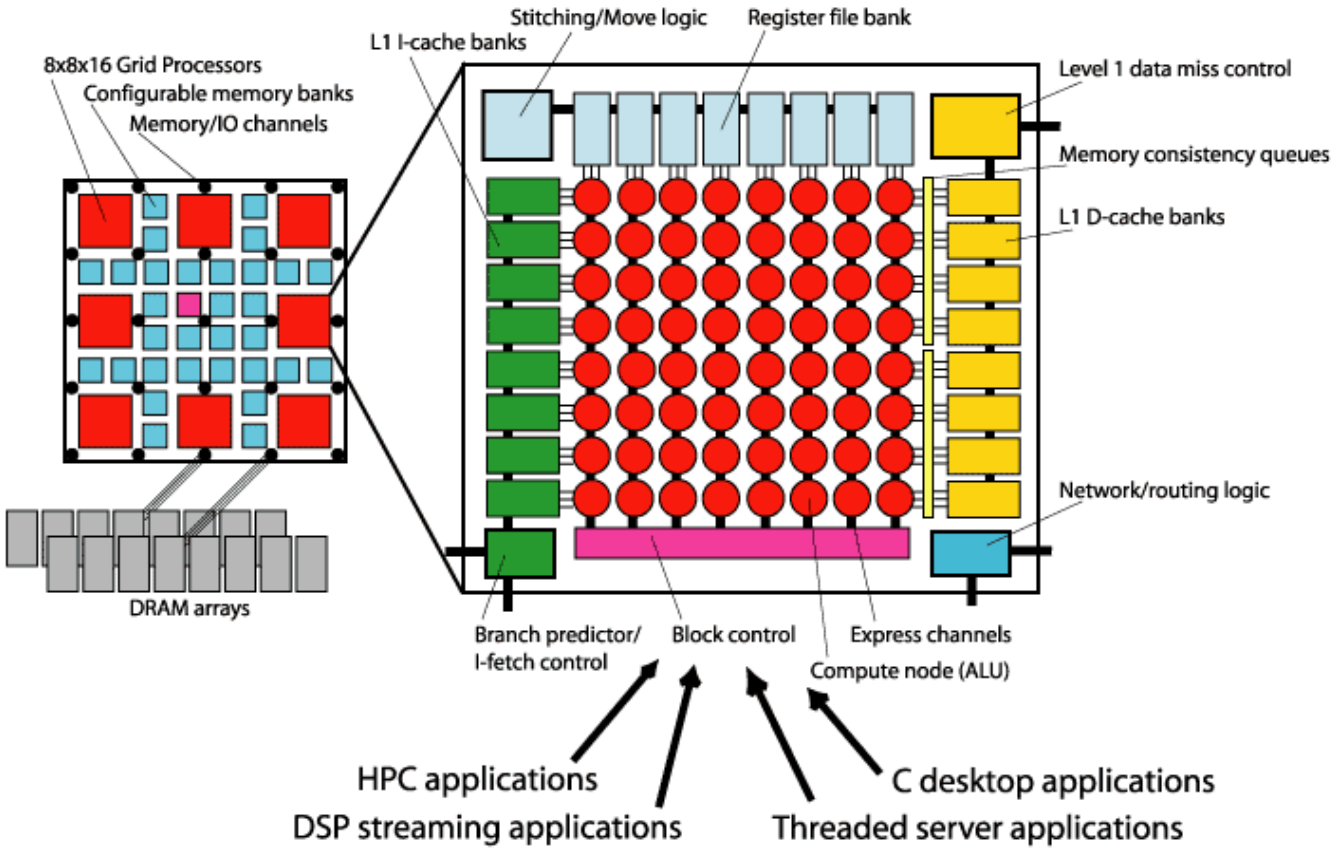
A Circuit Generator Theorem

We often generate netlists and their theorems. If we need to check an existing netlist, we may instead generate a netlist with the function we believe the existing netlist has and then use SAT to check their equivalence.

```
(defthm generate-ripple-se-rewrite
  (implies
   (generate-ripple-carry-& 4 netlist)
   (equal (se flg 'ripple-carry_4 params ins st env netlist)
          (let ((x (get-sublist (get-value flg ins env) 0 3))
                (y (get-sublist (get-value flg (cdr ins) env) 0 3))
                (c_in (get-sublist (get-value flg (cddr ins) env) 0 0)))
            (acl2-ripple-adder 4 x y c_in)))))
```

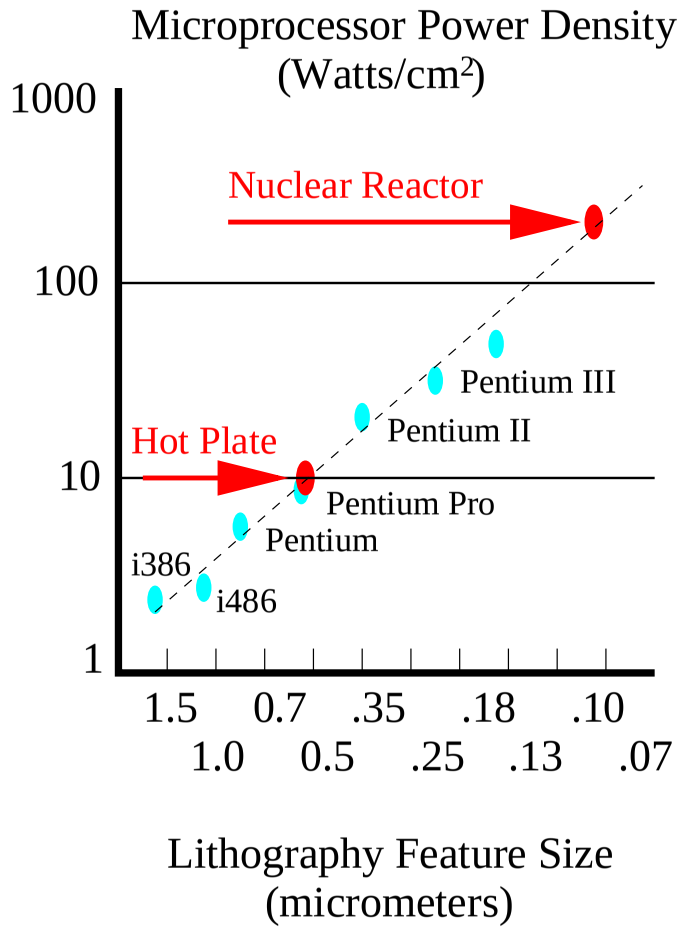
Our Stress Test

- We are using DE to design a part of the UT TRIPS architecture.



Combined Power & Functional Specifications

- The power density of microprocessors (as well as ASICs, FPGAs) is now first-order problem.
- There are ways to trade power for performance.
  - Equivalent circuits that require different numbers of clock cycles
  - Different circuits can be selected based on the current voltage.
- We are attempting to combine functional and power specifications into the DE language.
- Future functional verification will require knowing circuit parameters, such voltages, as well as the netlist.



Source: F. Pollack, Intel, New Microprocessor Challenges in Coming Generations of CMOS Technologies, Micro32

Our Vision

We would like computing systems to be specified by a *formula manual*, a complete precise set of formulas that exactly specifies computings systems (whether hardware, software, or both).

- A formula manual unambigously describes the functionality being offered.
- A formula manual defines the abstract specification for its concrete implementation.
- A formula manual can be used as the concrete specification for something built on top of it.

We want mathematically specified, mechanically checked computing systems.

- This is a long-term and evolving goal, `DE` is one such step.
  - Systems are increasing in complexity faster than our ability to manage them or control them.
  - If we are aggressive, maybe we can achieve this vision on small commercial designs, e.g, cell telephones, pagers, routers, etc.
- Our ability to field secure systems is based our our ability to specify and validate our computing, networking, and control systems.

Wired – a Language for Describing Non-Functional Properties of Digital Circuits

Emil Axelsson, Koen Claessen, Mary Sheeran

Chalmers University of Technology

{emax,koen,ms}@cs.chalmers.se

Abstract. Increasingly, designers need to estimate non-functional properties such as area, power consumption and timing, even when working at a high level of abstraction, early in the design. In deep sub-micron processes, it is the routing wires that account for most of the power consumption and signal delays. So, information about the wires is vital for controlling non-functional properties. To deal with more and more complex constructions, current design methods and languages strive towards higher and higher levels of abstraction, and provide only very limited possibilities for low level control. Often, detailed information about wire properties is only available in the very last design stages – after placement and routing. We propose a language, Wired, that aims to bridge this gap in abstraction levels. The main idea is the description of circuits using higher order connection patterns, or combinators. The key to the usefulness of this style is that the combinators have both functional and geometric interpretations. This allows us to construct circuits at a high level, without loosing control over lower levels. Wired goes further than previous methods in that it offers more precise and flexible control over the geometry.

1 Introduction

The enormously complex digital circuits that we see today – for example microprocessors with several million transistors – have been made possible due to a careful separation of abstraction levels. One of the most important abstractions is the one that views a chip full of transistors and wires as a netlist; a set of basic logic gates and their connections, completely without geometry. This makes it possible to focus on the circuit's behaviour and how to describe this correctly and efficiently, without being disturbed by the physical reality that is underneath.

In the traditional ASIC design-flow, the designer starts by describing the circuit in some high-level language, and uses a synthesis tool to create a netlist that realizes the same function. The netlist is then handed over to a place-and-route tool, which takes care of the non-functional synthesis – placement and routing of transistors and wires. This step can, obviously, not affect the circuit's function, but it certainly affects the non-functional behaviour, such as area,

power consumption and signal delays. Increasingly, it is the *routing wires* that contribute most to these non-functional properties. In earlier IC processes, wires were considered almost ideal connection elements, and the gates were the main contributors to non-functional effects. This made the link to the netlist (where gates are explicit and wires implicit) much clearer. But in today's deep submicron processes, the roles have been exchanged, and in fact, it is the wires that account for most of the the power consumption and signal delays. This problem is becoming more and more apparent as the development of new processes goes on, and will potentially soon become the major bottleneck for advancing circuit performance [1].

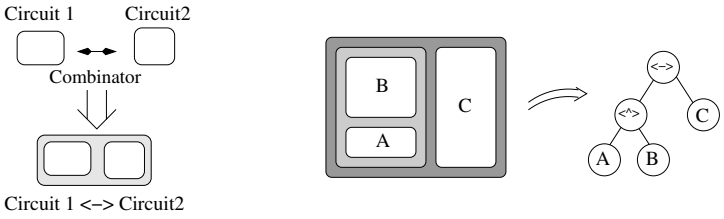
A good and compact layout will, naturally, result in shorter wires and smaller non-functional effects. Therefore, much effort is spent on making better place-and-route algorithms, an area in which there is still much work to be done. However, there is a fundamental problem with the way circuits are synthesized from high-level descriptions into layout. A circuit with a somewhat regular structure (usually true for arithmetic circuits) will most likely yield the most efficient layout if placement is done according to the inherent structure (see results from FPGA experiments in [2]). Although the structure may be well-known to the designer, there is no adequate way of communicating this knowledge to the lower synthesis levels.

This article presents a general language-based approach to efficient description of (somewhat) regular circuits, with layout as a natural ingredient. This is work in progress, and our first goal is to provide a framework for fast exploration of different layout alternatives. In the future, we also hope to include the language in a complete design-flow for data-path circuits.

2 The Wired concept

In Wired, circuits are built as compositions of smaller sub-circuits with so called *combinators*, a concept illustrated in figure 1 (a). This approach has previously been used in, for example, Lava and Ruby [3, 4]. A circuit described in this way is really a binary tree structure, where the leaves are primitive building blocks, and the internal nodes represent the different combinators. Figure 1 (b) shows a simple circuit and its binary tree representation.

This is indeed a very simple circuit model and may not seem very impressive at first glance. We will see, however, that its strength lies in the ability to describe several circuit aspects at the same time. We can think of a circuit as consisting of two parts – its functional and non-functional interpretations. The functional interpretation can be, in the most general case, a relation on boolean signals, and the non-functional can be the circuit's geometrical appearance. The combinators are defined for both of these aspects. In figure 2, we see a composition of circuits with two aspects. The result is a new circuit whose function is obtained by connecting some of the original circuits' signals, and whose geometrical appearance is obtained by placing the two original blocks next to each other. This



(a) (b)

Fig. 1. (a) Composition with combinator (b) Binary tree representation

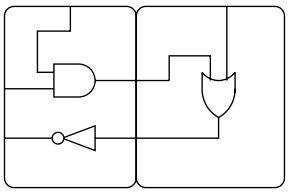


Fig. 2. Functional and non-functional composition

way of giving parallel properties to the circuits provides a semantically clear link between the functional and non-functional worlds.

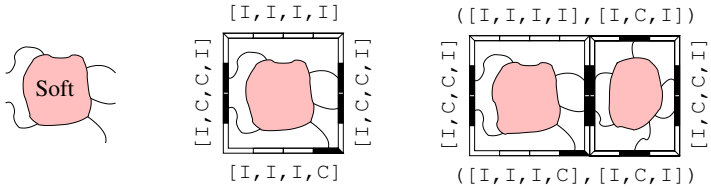
What distinguishes Wired from previous methods is that the circuits can be given a much more strict geometrical interpretation, and that wires are treated just like any other circuit. Since we don't abstract away from wires, this gives a new way to tackle the growing problem of wire properties being dominant. In a way, Wired can be seen as Lava with an extra non-functional property system added, and we believe in a design procedure where we start by making a description of the circuit in Lava, and when this is done, refine it to suit a description in Wired, and finally take the step into Wired by adding the non-functional information.

3 Wired circuits

We will now look more closely at the multi-purpose circuit descriptions mentioned earlier. We distinguish between circuit descriptions with only functional aspect – *soft circuits* – and those where the geometry aspect has been added – *hard circuits*. We will also assume that we already have some good representation of the soft circuits (for example as Lava descriptions), and instead concentrate on how to represent the geometry. The current chip technologies offer up to ten metal layers for routing wires, so in order to give an adequate model, we really need to think of circuits as being three-dimensional objects. In the future, we may also see real three-dimensional circuit technologies appearing [5]. Although a 3-D circuit model is our intention for the future, we have started by tackling

the simpler problem of describing 2-D circuits. In this simpler system we can explore our ideas, and when we are ready, the step to three dimensions should be straight-forward.

We can think of the soft circuit as a cloud with some signals going in and out, see figure 3 (a). A primitive hard circuit is then intuitively a soft circuit with a hard *frame* added, as in figure 3 (b). The frame consists of four ports, each one being a sequence of equally-spaced contact segments, possibly grouped into some structure. The contact segments, in turn, can be either *connections* (C) or *insulators* (I). Furthermore, there is a 1-1 mapping of the soft circuit's signals to the connections in the frame. The ports are named A, B, C and D, and the contact sequences are to be read in the direction indicated by the arrows in figure 3 (b). Finally, all frames are required to be rectangular.



(a) (b) (c)

Fig. 3. (a) Soft circuit (b) Primitive hard circuits (c) Compound hard circuit

To start with, we have two combinators for composing hard circuits – *'beside'* and *'below'*. In figure 3 (c), we see the result of 'beside' of two primitives – a new hard circuit whose two aspects are combinations of the original circuit's aspects. The functional behaviour is obtained from the original circuits by connecting some of their signals; however, it is actually the geometrical aspect that decides which signals should be connected – namely the ones whose associated C-contacts meet in the composition.

Wired is currently implemented in Haskell, and the frame properties are modelled as Haskell data type values. The contact property type is defined simply as

```
data ContProp = C | I | X
```

where `C` stands for connection, `I` for insulator and `X` for unknown. In defining the port property type, we will need different kinds of lists. We define

```
data List a = L [a] |
              R Int a
```

`L` is a list with known number of specified elements, and `R` is a number of repetitions of the same element. If the integer argument to `R` is negative, the number of repetitions is unknown. Now, we define the port property type as

```
data PortProp = Sim (List ContProp) |
                Com (List PortProp) |
                XP
```

According to this definition, a port property is either a simple sequence of contacts (`Sim`), a tree structure of port properties (`Com`), or unknown (`XP`). To simplify the expressions, we define the following helper functions, which we will use as simplified syntax for the port properties in the rest of the article:

```
sim cps   = Sim (L cps)
simr n cp = Sim (R n cp)
simx cp   = Sim (R (N (-1)) cp)

com pps   = Com (L pps)
comr n pp = Com (R n pp)
comx pp   = Com (R (N (-1)) pp)
```

For simple circuits, we will only need `sim` and `com`, the other ones will appear when we consider generic circuits in later sections.

Now we can define the frame properties of the hard circuits from figure 3. The primitives have simple structured lists:

```
   Circuit 1:                      Circuit 2:

Port A:  sim [I,C,C,I]          Port A:  sim [I,C,C,I]
Port B:  sim [I,I,I,I]          Port B:  sim [I,C,I]
Port C:  sim [I,I,I,C]          Port C:  sim [I,C,I]
Port D:  sim [I,C,C,I]          Port D:  sim [I,C,C,I]
```

But for the compound circuit, the B and C ports will be grouped into a pair, to reflect the fact that the port consists of two parts:

```
Port A: [I,C,C,I]
Port B: ([I,I,I,I], [I,C,I])
Port C: ([I,I,I,C], [I,C,I])
Port D: [I,C,C,I]
```

The port property can be seen as specifying the ports geometrical appearance, but also as defining the interface through which the port communicates. Therefore we have the requirement that *the properties of connecting ports must be equal*. This way it is checked that the circuits are used as intended – a C-contact is never connected to an I-contact, and a circuit that expects a pair of inputs is never connected to a triple etc.

Now, we can look at some simple circuit description examples in Wired. This article focuses mostly on how to describe the geometrical properties of circuits, and therefore we will leave out the internal representation. A hard primitive is created in Wired using the following function:

```
prim :: String ->
        (PortProp, PortProp, PortProp, PortProp) ->
        SoftCirc ->
        Circuit
prim name fProp sCirc = ...
```

The first two parameters define the non-functional aspect; `name` is a string that identifies the circuit, and `fProp` is the circuit's frame property, given as a quadruple `(ppA,ppB,ppC,ppD)`. And the last parameter, `sCirc`, is the soft circuit that we want to put inside the frame.

We can use this generator to create a small library of primitives to use in our constructions. For example a horizontal piece of wire with unit width and height can be defined as (assuming the existence of a soft circuit)

```
prim "wirH" (sim [C], sim [I], sim [I], sim [C]) wirHSoft
```

There are also operations for transforming the circuit's properties:

```
rot, flipH, flipV :: Circuit -> Circuit

rotN :: Int -> Circuit -> Circuit
```

`rot` rotates a circuit 90 degrees counter clockwise, `flipH` and `flipV` perform horizontal and vertical flipping, and `rotN n` applies `n` rotations to a circuit. So we can define a vertical wire as

```
wirV = rot wirH
```

Figure 4 shows some other simple wiring primitives, defined in a similar way. The number `0` at the end of some names is the rotation of the circuit. These also have versions with the numbers `1`, `2` and `3`, so that, for example

```
crT3 = rotN 3 crT0
```

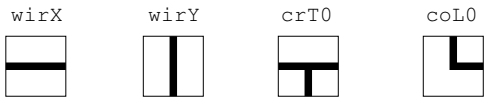


Fig. 4. Simple wiring primitives

A two input gate primitive, parameterized by soft circuit, can be defined as

```
gate2 softGate = prim "gate2"
                      (sim [C,C], sim [I,I], sim [I,I], sim [I,C])
                      softGate
```

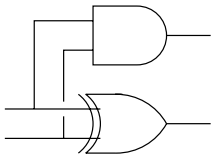


Fig. 5. Half adder

With this, we can define our standard gates: `and`, `nand`, `or`, `nor`, `xor`, etc. by just inserting the right soft circuit. For example:

```
and2 = gate2 and2Soft
```

Now we can start to compose these into the half adder circuit in figure 5, using the combinators `!>!` (beside), and `!^!` (below). The wiring on the input side is defined as

```
forkPair = ((crT2 !>! wirH) !^! (cro  !>! crT2))
             !^!
           ((wirV !>! coL3) !^! (coL3 !>! wirH))
```

and the gates are simply

```
gates = xor !^! and
```

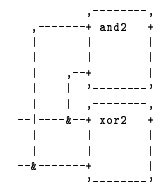
Before we compose these, let us look at the port properties of the two parts. This is done with the function `getFrameProps`:

```
  forkPair:

Port A:  com [com [sim [C], sim [C]], com [sim [I], sim L [I]]]
Port B:  com [sim [I], sim [I]]
Port C:  com [sim [I], sim [I]]
Port D:  com [com [sim [C], sim [C]], com [sim [C], sim L [C]]]

  gates:

Port A:  com [sim [C,C], sim [C,C]]
Port B:  sim [I,I]
Port C:  sim [I,I]
Port D:  com [sim [I,C], sim [I,C]]
```

We see that although the D port of `forkPair` and the A port of `gates` have the same contact sequence, the structures are slightly different, and thus they cannot be connected. However, there will be many cases where we want to allow different structures to be composed as long as the contact sequences match. For this, we have lax versions of the 'beside' and 'below' combinators: `.>.` and `.^.`. How these are implemented is seen in section 5. Now we can complete the half adder:

```
halfAdd = forkPair .>. gates
```

This is a complete Wired circuit. If we give it to the function `drawCirc`, we get the following ascii picture:



These ascii pictures are used as a sanity check of the constructed circuits, but other formats could easily be produced. We can also extract the Lava netlist from the circuits, to use for simulation, verification or VHDL generation.

4 Generic circuits

So far, we have only seen descriptions where all properties were completely defined, and all building blocks manually placed. We can get much more efficient descriptions with the use of *generic circuits*. In our descriptions we will be needing straight wires in many different lengths. Instead of defining different circuits for all of these, we can have the size as a circuit parameter:

```
wirHS s = prim "wirHS" (sim [C], simr s I, simr s I, sim [C]) ...
```

Here we used `simr` to describe the port properties as repetitions instead of the explicit contact lists we used before. This adds a level to the expressiveness of the language, but we will still be limited by the fact that we have to know all wire sizes before constructing our circuits. For this reason we have generic port properties – properties containing any of `X`, `XP` or `simr/comr (-1)`. A generic port gets instantiated from the context into which it is placed, a direct consequence of the requirement of connecting ports having equal properties. For example, if in a composition, an X-contact connects to a C-contact, the X will be instantiated to a C. Similarly, we can have circuits with unspecified length that adapt properly to the circuits around. A first attempt to a generic-length wire could be

```
wirX = wirHS (-1)
```

This will work if the context has known length on both sides of the wire, but if only one side is known, only one of the ports will be instantiated. Earlier, we stated that all frames must be rectangular, a requirement that can be used to propagate size information across circuits. To do this, we define `wirX` as a generic circuit:

```
wirX = generic "wirX" (sim [C], simr (-1) I, simr (-1) I, sim [C]) wirXI
```

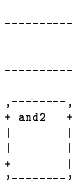
This looks like a definition of a primitive, but instead of the soft circuit, we have an *instantiation function*, `wirXI`. This function defines, for all possible contexts, a new updated version of the circuit. Although this leaves much room for experimentation, the instantiation function usually has the following structure:

```
wirXI (ppA,ppB,ppC,ppD)  | (len >= 0) = wirHS len
                         | otherwise  = wirX
  where len = max (sizePP ppB) (sizePP ppC)
```

Here we use `sizePP` to test the context properties. This function returns the size of a port property in number of contacts, or (-1) for ports with unknown length. Thus, if either of port B or port C has known length, the instantiation will take the first case and result in a `wirHS` with known length. If both ports have unknown length, we get back the original `wirX` circuit. This last case is always needed, because the instantiation algorithm needs several iterations to solve the properties of complex circuits. Now we should be able to get a correct instantiation of the following circuit:

```
testCirc = and2 !^! wirX !^! wirX
```

`drawCirc` gives:



which shows that size information has been propagated from the and-gate, through the middle wire to the top wire.

5 Port maps and structure conversion

We saw in the half adder example that it is very likely that we will need to connect circuits whose port structures don't match exactly. In such cases, we use something called *port maps*. A port map is really nothing but a thin circuit, see figure 6 (a). The top and bottom properties of the port map are empty lists `sim []`, and since it is infinitely thin, it must have identical contact sequences on both sides. But the important thing about it is that the structures of the side properties can be completely different! This means that we can actually use port maps as property structure converters. So, by placing proper port maps in between, we can now compose circuits where the structures don't match, as long as the contact sequences do. The only problem with this is that though the port map really is a circuit, we cannot compose it the way circuits are normally composed. The reason for this is that it would mess up the orthogonal ports of the resulting circuit, see figure 6 (b). The intention of the port map was only

to convert one of the port properties, not to touch the others. Therefore we say that port maps are *applied* to circuits, rather than composed. This is done with four new combinators: `apmA`, `apmB`, `apmC`, and `apmD`, for port map application to port A, B, C and D respectively. The effect of port map application is shown in figure 6 (c).

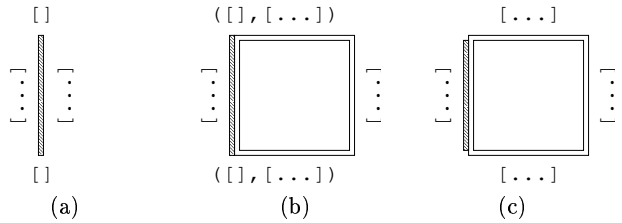


Fig. 6. (a) Port map – a thin circuit (b) Port map *composition* c) Port map *application*

A simple port map is generated by the function

```
portMap :: String -> (PortProp, PortProp) -> Circuit
portMap name (ppA,ppD) = ...
```

It only needs the A and D properties, B and C will become `sim []`. This really defines a generic circuit similar to `wirX`, but instead of just sharing size between the opposite ports, the port map also shares the contacts.

The most general port map is obtained as

```
genPM = portMap "genPM" (XP,XP)
```

It can be used to implement the lax combinators we saw earlier:

```
circ1 .>. circ2 = (apmD genPM circ1) !>! circ2

circ1 .^. circ2 = (apmB genPM circ1) !^! circ2
```

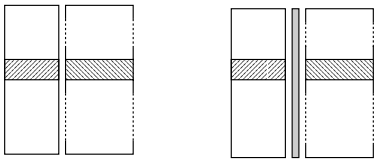
Another useful port map is the one that converts any property structure to a simple one:

```
toSimPM = portMap "toSimPM" (XP, simx X)
```

A function that converts all ports of a circuit to simple ones is

```
allToSim = (apmD toSimPM).(apmC toSimPM).(apmB toSimPM).(apmA toSimPM)
```

Even if the port maps seem to be very useful, they should be used with care when there are generic circuits involved. Consider the composition in figure 7 (a). The left component has fixed size, but the right consists of two generic parts. What happens when these are composed is that since the structures match (both



(a) (b)

Fig. 7. (a) Possible to instantiate? (b) With a port map in between ?

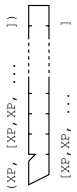


Fig. 8. The `cons` port map

are triples), the three parts will be instantiated one by one, and thus the circuit is possible to instantiate. But in figure 7 (b), we have the same circuit with a port map in between. In that case, we have spoiled the possibility of doing structure matching, and the circuit cannot (generally) be instantiated.

Finally, a port map that will be used in the next section, is the `cons`, illustrated in figure 8. The intuition behind `cons` is that it takes a list and splits it into a pair, consisting of the first element and the rest. Its generic port properties are:

```
Port A:  com [XP, comx XP]
Port D:  comx XP
```

For this we cannot use the `portMap` function, because it is not enough that the contacts are shared between the sides, we also want the list elements (the different `XP`'s) to be shared.

6 Connection patterns

In this section we will look at another way to build generic circuits, *connection patterns*. This is really the key to efficient description of larger circuits, as they are used to generically encode the structure of regular circuits, independent of size. The patterns define recursive structures where the same circuit is repeated over and over again, and the repeated circuit (or part of it) is given as a parameter. The most basic one is the *row*, as illustrated in figure 9.

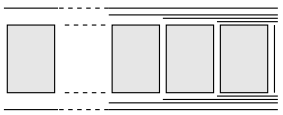


Fig. 9. row connection pattern

It can be defined as a generic circuit using the previously defined `cons` port map (details are left out):

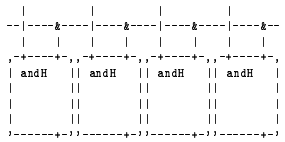
```
row circ = generic "row" (XP, comx XP, comx XP, XP) (rowI circ)

rowI circ ppQuad
   | (contextSize == -1) = row circ
   | (contextSize == 0)  = thinID
   | otherwise           = ((apmB cons).(apmC cons)) (circ !>! row circ)
  where ...
```

The interesting part here is the instantiation function and its three cases. When the row's context is unknown, it continues to be just a `row circ`. But when the context size is known and greater than zero, it instantiates to one copy of `circ` next to another row (the recursive case). As the instantiation proceeds, the context eventually reaches the size zero. Then it instantiates to a single `thinID`, which is a thin circuit with identical properties on its both sides (the base case). The recursive case is wrapped in `cons` portmaps to keep the structures on the top and the bottom clean. The effect of instantiating such a row is seen in figure 9 (b).

The row can be used for building all kinds of linear structures, such as registers, ripple-carry adders etc. Here is an example of a simple bit multiplier circuit. We build it from and-gates with a special wiring that reads the multiplicand bits from the top, and the multiplier bit from the left. It also passes the multiplier bit on as a carry to the right:

```
and_bitM = (rotN 3 and2) .^. (cro !>! crT0)
```

The bit multiplier is then simply `row and_bitM`. Instantiation for four bits gives:



7 Generalized property system

In this section we will give a brief outline of a more generalized property system, which will extend the language's expressiveness even further. As before, the circuits have two aspects, functional and non-functional, but the non-functional part can now be any set of properties. A non-functional property in this sense could be: a circuit name, a hard frame (two- or three-dimensional) or a soft frame (in our current implementation, the soft circuits are frame-less). We will also have a very general instantiation algorithm that works for any number of combinators, and circuits with any kinds of properties. The effect each combinator has on the different properties is specified separately.

We can think of many advantages of this system:

- It will make the transition to three-dimensional circuits smooth and reliable.
- It makes it possible to mix different property systems in different parts of the circuit. For example, flat circuit parts could be described two-dimensionally and later be converted to 3D. Or if we want to connect two circuits quickly without having to describe a complex wiring network between them, we could just convert them to soft circuits (wrap them into a cloud) and connect them with a simpler soft combinator. In fact, a larger design could contain a soft-hard mix, where the easily laid out parts are hard circuits and the others are soft. The soft parts would then have to be implemented by an external place-and-route tool.
- It is also possible for the user to add his own property types, and to define his own combinators with specialized semantics. For example, a user-defined property could be a list of integers representing the delay of each individual signal to a component. This information is then propagated through the circuit during generation as a guide for different instantiation alternatives for generic circuits.

The extra properties can intuitively be thought of as shadow information described parallel to the circuit. The idea of using shadow information to guide circuit generation has already been shown to work well in Lava [6]. Its combination with a generalized property system promises to be a powerful design aid.

8 Types

In the current implementation, errors due to misused properties will be detected at circuit generation time, or later. With a type system, even a simple one, we could probably detect many of these errors one step earlier. The current property system was actually first intended as a type system for hard circuits. But in order to make it modular, we had to lift all instantiation mechanisms to the type level, and eventually this became too complex to handle. In the future, we will try to make a much simpler type system, concentrating on signal structure instead of geometry.

9 Conclusions

We have presented a language for describing functional and non-functional aspects of circuits in parallel. This seems a promising approach to tackle the growing problem of wire effects taking over the performance in deep sub-micron technologies. The expressiveness is increased by the use of generic circuits that adapt to their context, and connection patterns that efficiently encode the generic structure of regular circuits. Thus, regular circuits, like data-paths, will be a typical application area of the language. In this article, we have only seen very simple examples, but we believe that more complex patterns like different trees, and almost regular structures can also be efficiently described.

10 Related work

Description with combinators and parallel functional and non-functional interpretations has previously been used in languages like Hydra [7], Lava and Ruby.

Cong et al have developed an "Interconnect-centric design-flow" [8], which is a complete design-flow where wires are taken into account right from the start. Much of their focus is on making accurate early predictions of performance, but also on synthesis algorithms that properly take wire effects into account.

There has also been recent work on synthesizing efficient arithmetic circuits, taking account of gate and wire delays [9, 10].

However, we don't know of any related work where a language-based approach is taken to describe circuits with layout and explicit wires.

11 Future work

We want to continue to describe many more example circuits, to see how far we can get with the current system, and what can be done better. We are currently working on a general tree structure where many different multipliers can be obtained by just replacing some of the wiring. In order to get feedback on the effectiveness of the layout, we want to make methods for estimating different non-functional properties. To begin with, this will aim at wire length estimations, which can then be used for more accurate estimations of signal delays. The step after that is to see how we can interact with layout tools in order to make real constructions.

We will also complete the generalized property system, which will lay the foundation for our ideas of for example, 3D circuits, soft-hard mixing, and shadow properties. A future application of this system may also be hierarchical circuit verification.

On the language implementation side, we want to include a type system for early detection of property mismatches. And also, further ahead, we would like to make Wired a language on its own, apart from Haskell.

12 Acknowledgement

This research is funded by the Semiconductor Research Corporation, in an Intel-custom research project called Expressing and Estimating Non-Functional Properties of Digital Circuits (Task id. 1041.001).

References

1. D. Sylvester and K. Keutzer: Getting to the Bottom of Deep Submicron. Proc. ICCAD, 1998, pp. 203-211.
2. Satnam Singh: Death of the RLOC? FCCM'00. IEEE Computer Society. 2000.
3. K. Claessen, M. Sheeran and S. Singh: The design and verification of a sorter core. Proceedings of the 11th Advanced Working Conference on Correct Hardware Design and Verification Methods, vol. 2144 of LNCS, Springer-Verlag, 2001.
4. G. Jones and M. Sheeran: Circuit design in Ruby. In Formal Methods for VLSI Design, J. Staunstrup ed., North-Holland, 1990.
5. Matrix Semiconductor Inc., http://www.matrixsemi.com
6. M. Sheeran: Finding regularity: describing and analysing circuits that are not quite regular. Proceedings 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, vol. 2860 of LNCS, Springer-Verlag, 2003.
7. John T O'Donnell: Hardware description with recursion equations. In M R Barbacci and C J Koomen, editors, CHDL 87: 8th International Symposium on Computer Hardware Description Languages and Their Applications, pages 363–382. IFIP WG 10.2, North Holland, 1987.
8. J. Cong: An interconnect-centric design flow for nanometer technologies. Proceedings of Int'l Symp. on VLSI Technology, Systems, and Applications, pp. 54–57, June, 1999.
9. Junhyung Um and Taewhan Kim: Layout-Aware Synthesis of Arithmetic Circuits. Proceedings Design Automation Conference, 2002.
10. C. Martel, V. Oklobdzija, R. Ravi, and P. Stelling: Design strategies for optimal-multiplier circuits. Proceedings 12th Symposium on Computer Arithmetic, 12:42–49, 1995.

Integrating Formal Methods with Digital Circuit Design in Hydra

John O'Donnell *

Circuit design often takes place in two distinct phases. First the circuit is designed informally to meet a specification. The design may be expressed using a computer hardware description language (CHDL) for precise description of its structure and behaviour, as well as simulation. Then, once the design is completed, it is checked formally to verify that it meets the specification, using techniques such as theorem proving and model checking.

When we separate the design process from the application of formal methods, we miss some of the potential benefits. Formal and semi-formal reasoning can help the designer to find a way to implement the circuit, as well as improving confidence in its correctness. Furthermore, a circuit designed along with its correctness proof may end up with a structure better suited for verification.

This talk describes how formal reasoning can be applied during the design process using Hydra, a CHDL implemented by embedding within Template Haskell. Hydra specifications may describe the behaviour of a circuit, its structure, or both. Reasoning about the behaviour of a circuit is done in the context of a circuit model, which states what aspects of the circuit are encompassed (e.g. logic values and clock cycles) and which aspects are abstracted out (e.g. gate delays and glitches). Hydra supports several circuit models, allowing a choice of the level of detail.

Hydra represents various aspects of a circuit using Haskell type classes. In particular, the circuit structure is obtained by signal instances which build a graph that is isomorphic to the circuit. Traversal of the graph is problematical in a pure functional language. This issue is solved by an automated program transformation that converts the specification written by the circuit designer into a form where signals have unique labels, enabling the circuit structure graphs to be traversed without recourse to impure language features. In this way, the soundness of equational reasoning on Hydra specifications is uncompromised, yet circuit graphs with feedback can be traversed in order to generate netlists, produce representations to be used by formal verifiers, and so on.

Hydra began as an application of the work by Steven Johnson on hardware description using stream recursion equations, and was also influenced by Mary Sheeran's muFP language. It evolved via lazy evaluation: Hydra was used for a variety of circuit designs and circuit proofs, and those projects highlighted areas that needed better CHDL and tool support, leading to a number of improved techniques. Some of the early innovations in Hydra were multiple circuit semantics, integration of specialised sublanguages (e.g. to support control circuit design), the extension of fold and scan combinators into a flexible set of layout combinators, and the use of graph traversal with pointer equality to generate netlists.

The most important recent innovation in Hydra has been a new approach to the problem of generating netlists, using metaprogramming with Template Haskell. This makes it possible to operate in a pure functional language, with equational reasoning on a completely sound basis, while at the same time being able to generate netlists and to offer a variety of useful testing, simulation and debugging tools.

The talk will discuss the foundation of formal reasoning in Hydra (safe equational reasoning while still keeping the ability to generate netlists), as well as experience with designing circuits along with proofs of some of their properties.

*Author's address: Computing Science Department, University of Glasgow. jtod@dcs.gla.ac.uk

HML: A language for high-level design of high-frequency circuits

Andrew K. Martin

Feb. 16, 2004

Abstract

The last twenty years have seen large advances in the field of programming languages for software development. New languages such as Java, C++, and ML have largely replaced older, less sophisticated programming languages such as Algol, Pascal, C, and Fortran for new software development. These new languages have sophisticated data modeling and representation systems that hold out the promise of increased programmer productivity by permitting the design of libraries that build common layers of abstraction over implementation details. Programmers can then work effectively at these higher levels of abstraction, while still producing efficient code. Nonetheless, much performance critical code is still written in Fortran, C, or even assembler.

Commercially used hardware design languages are extremely primitive, requiring designers to express their designs at a very low level of abstraction, in relatively verbose languages such as VHDL and Verilog. Although these languages have, in some cases quite sophisticated "behavioral" representations, such representations are generally avoided because mechanical mechanisms to translate from the high level of abstraction to a low-level implementation are not thought to yield optimal implementations. In spite of years of research in "high-level synthesis", designers still choose to work at practically the gate level.

In high-performance microprocessor designs, this desire to work at the gate level stems largely from the need to maximize the performance of the resulting implementation. If target frequencies continue to increase more rapidly than the speed of the underlying process technologies, this need to maximize performance will be even more pressing. Nonetheless, working at higher-levels of abstraction could have many advantages for hardware design. Many classes of errors could be avoided completely. Others, that now must be detected by low-level simulation, could be detected by static analysis such as type-checking during the compilation process. Moreover, when a design has a "clean" abstract representation, it is easier to partition it into components, enabling more comprehensive verification using formal and traditional approaches.

To allow the designers of high frequency devices to take advantage of more abstract representations without sacrificing performance, we envisage the following design process. First, a design is expressed at the mircoarchitecture level of abstraction. Then the design space within the given micro-architecture is explored iteratively through a series of mappings that transform the high-level design representation into a low-level implementation. The designer writes an explicit map from the high-level representation to an implementation. A compiler then applies the map to the high-level design to produce a low-level (gate or switch) representation. This result is then analyzed using standard tools for timing, area, power etc. This information is then used by the designer to adjust the mapping in an iterative process. The expectation is that making the map from high-level representation to implementation explicit will give the designer sufficient control of the synthesis process that even designers of high-frequency components will be able to achieve their performance targets without resorting to designing at the gate level.

There are three main components to this research effort. A suitable high-level design language must be devised that allows for the rapid expression of designs at the micro-architecture level. A notation must be devised for expressing the mappings from high-level design to low-level implementation. Finally a suitable format for presenting the results of the mapped design must be devised, so that the results can be interpreted in a meaningful way by the designer, and hence guide the search for an optimal implementation of the original design.

This presentation will discuss the first step in this research direction, the hardware design language HML. HML is a hardware description language based on the OCaml dialect of the ML programming language. We have used it successfully at IBM's Austin Research Laboratory in the design of a recent test-chip. The use of HML allowed a very small team to design and integrate the chip in a period of less than four months. Logic design and verification was performed by only three people in this time-frame. The definition of HML is still somewhat fluid. Since, at present it is used only in a small research group, features can be added or removed from the language as dictated by our experience, both in using the language, and implementing tools that support it. The presentation will describe one snapshot of the language as used to design a recent test-chip.

HML was designed by the author at IBM's Austin Research Laboratory. HML augments ML with a polymorphic primitive *stream* type, to represent the sequence of values that occur over time on a signal in a synchronous design. Hardware components are represented as functions from input streams to output streams. Component instantiation is represented by function application. As in ML, HML treats functions as first class values: They can be passed as parameters, to other functions, returned as results, and constructed anonymously. Since hardware components are represented as functions, they can be passed as arguments to functions, or to other hardware components, returned as results etc. To describe a piece of hardware in HML one simply writes a function.

The compiler performs HAWK-style *lifting* automatically. Functions that were not initially designed to work with streams are automatically promoted by the compiler to functions over streams as required. For example, the function `fun x -> not x` can be applied to a boolean argument, which it will complement, or it can be applied to a *bool stream* in which case it will return a stream of booleans – the point-wise complementation of its. State is represented in HML by the syntax `delay(v1,v2)` where $v1$ is a value of type a, and $v2$ is a a-stream. The result is a stream whose first value is $v1$, and whose subsequent values are those of $v2$ (delayed one unit of time).

The type system includes the atomic types "int", "bool", and "unit". Constructed types include heterogeneous tuples, homogeneous arrays, records with named fields and variants – type-safe tagged unions. The type system departs from that of OCAML, by introducing a type hierarchy, in which sub-types can be used any place where a super-type is expected. This allows one to use a record of type `{a: a_type1; b: b_type1; c: c_type1}` where a record of type `{a: a_type2; c: c_type2}` is expected, provided that `a_type1` is a sub-type of `a_type2` and `c_type1` is a sub-type of `c_type2`. The converse sub-typing rule applies to variants, so that a variant with less constructors can be used any place where a variant with more constructors can be handled. Records and variants are co-variant in their components, while function types are co-variant in their result type, but contra-variant in their argument types.

HML derives much of it's basic syntactic form from ML. An HML program is an expression which evaluates to a function from streams to streams. Like OCaml, HML allows pattern matching in both let-bindings and formal parameter declarations. For example `let (a,b,c) = e1 in e2` evacuates $e2$ in an environment in which "a", "b", and "c" are bound to the components of the triple e1. However, HML extends the Ocaml pattern matching to destruct record types. For example `let {a;b} = e1 in e2` evaluates $e2$ in an environment in which "a" and "b" are bound respectively to the "a" and "b" fields of e1, which must be a record type with at least the fields "a" and "b". The traditional array indexing syntax from Ocaml (`a.(i)` refers to the "i^{th}" element of the array a) is replaced by the syntax `a[i]`, and extended to include the syntax `e1 [ e2 : e3]` to extract an array subrange.

The ability to destruct records "on the fly" in formal parameter definitions and "let" expressions provides a lite-weight mechanism for binding function parameters by name. Thus a function "foo" can be defined with three arguments "a", "b", and "c" that are bound by position: `let foo a b c = e` and called using the syntax `foo av bv cv`. For functions with many arguments of the same type, it is very easy to for a programmer to err by providing the arguments in the wrong order. Since large hardware components often have many hundreds of incoming signals, hardware description languages usually offer a bind-by-name mechanism for module instantiation. The same effect can be obtained in HML using record types. Thus, the same function could be defined using bind-by-name: `foo {a=a;b=b;c=c}` The order in which the fields are listed, both in the formal parameter definition, and in the actual parameter is unimportant.

For the test-chip an HML to net-list compiler was built that implemented a highly restricted subset of the language. No "synthesis" was supported in the usual sense. Instead, a library of primitive functions with hardware implementations was provided to the compiler. The resulting net-list consisted entirely of instantiations of these primitive circuit elements. Streams were restricted types whose atomic elements were booleans – thus integer, record, array, and function valued signals were disallowed.

The original implementation, built in a matter of weeks, was based upon graph reduction. HML programs are translated into a combinator graph, which is then reduced. Net-lists were created as a side-effect of graph reduction. While this seemed, initially, like a simple, elegant approach to compilation, in retrospect it was not. There were numerous times when a device with no outputs, or several devices with no inputs needed to be instantiated. This required a sequential operator whose first argument is evaluated strictly for it's side-effects. Soon, the "lazy" evaluation semantics of the combinator graph, had to be enhanced with an "eager" evaluation semantics. Future implementations will move to more conventional compilation technologies, in which the resulting net-list is the product of an explicit translation, followed by elaboration, rather than the side-effect of a partial evaluation.

The implementation time frame precluded the design of a static type checking and type inferencing mechanism. As a result, all type-checking was done a run-time (roughly analogous to elaboration time for traditional languages). Indeed, even type annotations, while allowed syntactically, were not supported. It turns out support for explicit type annotation is even more critical in a hardware description language than in a programming language. While static type inference (or in the case of our first implementation, dynamic type checking) is very convenient at the beginning of the development process. Since types are rarely written explicitly the design can be changed rapidly without having to change type-definitions or symbol declarations. However, late in the design process, this extreme flexibility ceases to become an asset, and becomes a liability. Late in the process, once physical design has begin, it is critically important that the interfaces to components that correspond to physical entities on the chip are not changed inadvertently. The ability to annotate interfaces with type specifications would have been a great improvement, which will certainly be incorporated into future compilers.

In spite of these restrictions and issues, the initial implementation turned out to have considerable value. A substantial test-chip, intended to test a novel digital MOS circuit family designed to run at very high-frequencies, was fabricated in mid 2003. The design of the entire chip, with the exception of the PLL and clock control circuitry, was expressed in HML. It was mechanically translated to VHDL for functional verification and to a net-list for layout. In all, about 1500 lines of HML code were translated into 331,000 lines of VHDL, of which 133,000 were instantiated decoupling capacitor cells, and the remaining 198,000 lines were a structural VHDL description of functional circuit elements. The use of HML allowed a very small team to design and integrate the chip in a period of less than four months. Logic design and verification was performed by only three people in this time-frame.

HML represents a first step in an efficient system for logic design for high frequency circuits. The compiler used in the test site was essentially a net-lister. That is, the designer writes an ML program that is interpreted as a set of cell instantiations. Work is currently under way to build a second version of the compiler, which will address the issues noted above, while providing more comprehensive simulation and synthesis capabilities. In particular the second two research problems: a notation to express mappings from the high-level design to implementation, and a format for presenting the mapped design have yet to be addressed.

Late Design Changes (ECOs) for sequentially optimized high-level Esterel designs

Laurent Arditi, Gerard Berry
Esterel Technologies
{Laurent.Arditi,Gerard.Berry}@esterel-technologies.com

Michael Kishinevsky
Intel Strategic CAD Labs
Michael.Kishinevsky@intel.com

February 20, 2004

1 Introduction

Late changes in silicon design, called ECO, is a common although undesired practice. They happen due to last minute changes in the specifications or to design bugs found at the late stage, sometimes after the tapeout. At this stages going through the top-down design flow is infeasible, because it would take too long and lead to undesirably large perturbations to the physical layout. High-level design flows generally reduce the number of potential bugs. However, the need for ECO still exists since there is no guarantee that all bugs are eliminated and since the spec may change late in the game. Since high-level design often deploys more powerful optimization than manual design flows, it becomes harder to find the place in the final circuit where manual changes should be done in order to correct the behavior. It is also harder to trace circuit bugs and changes back to the high-level spec. A software analogy would be to of perform manual changes in a C executable compiled with -Ox options, while back-annotating these changes to the original C code.

We will illustrate this general high-level design problem by an Esterel example, with heavy sequential circuit optimization performed by the Esterel compiler backend. The desired ECO flow is as follows. The original specification S is compiled by the Esterel compiler to a circuit netlist C_0, which is further optimized to the final implementation C using combinational and sequential optimization methods. If late changes are required this circuit is transformed manually into another circuit netlist C^* such that perturbations to C are minimal. To maintain the high-level specification consistent with the modified implementation and to verify the manual change to the implementation, the original Esterel specification S is also modified into S' to reflect the late changes. S' is then compiled to a new netlist C'. Finally, C^* is verified against C'. To debug mismatches, it is necessary to understand if the fault is in the manual changes to the implementation or in the changes to the Esterel spec, iterating until perfect match. The circuit C' can be used exclusively for verification or to provide hints for modifications of the original implementation C. In the first case, optimization of C' is optional but can speed up

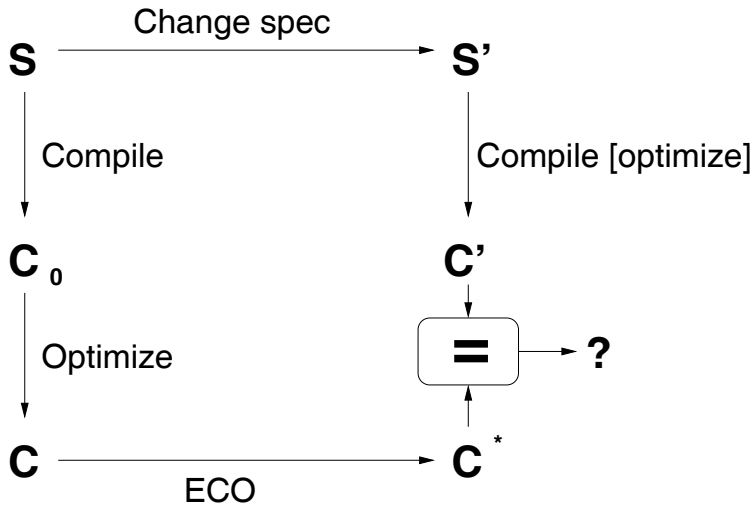


Figure 1: ECO flow

verification. The ECO flow is illustrated in Figure 1.

Three capabilities are required for the above ECO flow:

- *A sequential equivalence checker*. Since sequential optimization is involved into producing C and possibly C', the circuits C^* and C' are in general not combinationally equivalent. Esterel Studio has two embedded sequential equivalence checking engines that are based on BDD [5] and SAT [8]. Capacity of these tools is typically matching capacity of sequential optimization, and it is rarely an issue in the modular compilation flow for control-dominated designs. Recent work demonstrated that capacity of sequential verification can be further improved, e.g., by using combination of ATPG (or SAT) and BDD approaches [6], structural equivalence [14], and multiplexing the state of the FSMs under verification [7]. More research in this direction is required to support high-level design flows.

- *Tracability tools* to zoom into the parts of the implementation netlist C, where manual changes corresponding to the modified spec S' should be done. Tracability has been implemented in the Esterel Studio 5.0.1. It supports forward linking of every source construct (state, transition, textual Esterel statement) with HDL objects (variables, signals, logic equations) and backward linking of any generated HDL object to its Esterel source. Traceability is presented in Section 2. Notice that traceability is also the basis of critical software certification flows, see [].

- *Modular compilation*, necessary to confine changes to relatively small circuit blocks. The next version of Esterel Studio will support modular compilation, with minor reincanation limitations (a separately compiled module will not be able to support multiple calls in the same cycle). The user will be able to control the grouping of modules, and hence to choose the granularity at which to optimize the design and perform ECOs.

Sequential optimization used in the Esterel backend includes a few transformations: redundant latch removal, reencoding of exclusive latches, retiming moves, reencoding of sequential threads,

and code migration [10]. We distinguish between reversible and irreversible optimization transformations. In a reversible transformation, one can reconstruct removed registers from the registers of the final circuit and some extra information kept inside the design. All transformations that are bijective or injective on the state space of a circuit are reversible. Surjective transformations (like state minimization) are irreversible: if two states are collapsed they cannot be separated back.

Exlpoiting reversibility, we will show how an ECO problem for the reversible sequential transformations can be reduced to a combinational one by reconstructing some of the suppressed egisters in order to backannotate to the original code and to perform division between the logic of an actual design and the logic supporting backannotation. The combinational ECO problem is solved in standard design practice and is supported by some automation (e.g. by the ECO compiler of Synopsys).

The rest of the paper is organized as follows. Section 2 presents basics of the Esterel compilation flow and explains how tracebility is supported. Section 3 discusses interaction of sequential optimization with the ECO. ECO on examples is presented in Sections 4 for the unoptimized case and 5 for the optimized case. We conclude in Section 6.

2 Traceability in Esterel v7 compiler

2.1 A basic Esterel example

We illustrate the traceability between the Esterel v7 source code and the generated circuit using the following simple program:

```
main module Main :
input A, B;
output X, Y;
abort
   await A;
   emit X
when B;
sustain Y
end module
```

2.2 First step: building Esterel assembly code

The above program is first translated into an intermediate esterel assembly code, yielding (approximately) the following sequence of instructions:

```
statements: 9
0: 0 Root: (4) %lc: 0 1 1%
1: 0 Present: [B] (7, 2 %lc: 0 4 1%) %lc: 0 7 1%
2: 0 Resume: <6> %lc: 0 4 1%
3: 0 Present: [A] (5 ,4 %lc: 0 5 4%) %lc: 0 5 4%
4: 0 Pause: (3) <2> %lc: 0 5 4%
5: 0 Emit: [X] (7) %lc: 0 6 4%
6: 0 Watch: {1} <0> %lc: 0 4 1%
7: 0 Emit: [Y] (8) %lc: 0 8 1%
8: 0 Pause: (7) <0> %lc: 0 9 1%
```

The statements are indexed. The number after the index is a module instance index telling by which module instantiation the statement is generated. Here, there is only one module and all indices are 0. Pragmas such as `%lc: 0 5 4%` are source code backannotations, telling that a statement or part of a statement is generated from line 5, column 4 of file indexed 0, pointing here to the `await` keyword.

The flow of control is fully explicit, making statement ordering irrelevant. Direct continuations always appear between parentheses, while statement references between angle brackets serve for statement resumptions as explained below.

The `Root` instruction is the start point, with immediate continuation the `Pause` statement 4. Therefore, when the program is started, it immediately pauses at 4. Resumption from this point at next tick is based on the indices between angle brackets, which determine the selection father of a statement, as explained in full details in [2, 3]. Here, the `Pause` statement 4 signals that is is alive to its `Resume` father 2, which itself signals that it is alive to its `Watch` abortion father 6, which in turn signals aliveness to the `Root` statement. Resumption follows the reverse path. The `Root` statement resumes its `Watch` son, which implements the `abort` behavior by immediately triggering its associated `Present` test 1 whose index is given between curly brackets. The `Present` statement conditionally triggers the following behaviors:

- If `B` is present, the continuation 7 is taken, `Y` is emitted, and the program pauses at 8.
- If `B` is absent, control is transferred to the `Resume` statement, which resumes its son 4. Resumption of the Pause statement passes control to its continuation 3, which tests for `A`. If `A` is present, control is passed to 5, which emits `X`, and then to 7 to emit `Y` and pause. If `A` is absent, control comes back to the Pause statement that pauses again until the next instant, realizing the "`await A`" behavior.

Notice that the selection / resumption mechanism ensures the right priority between the tests for `A` and `B`. The backannotation pragmas are used to animate the source code in the Esterel Studio GUI.

The assembly code is notably more complex for full-fledged programs, with `Fork` and `Parallel` (join) statements to deal with synchronous concurrency and `Exit` statements to deal with the `trap-exit` Esterel exception mechanism. We give no more details here, since the handling of all statements is pretty similar as far as traceability and ECOs are concerned.

2.2.1 Second step: translation to circuit

From the above generated Esterel assembly code, the Esterel v7 compiler generates the circuit pictured in Figure 2. The gate names are abbreviated in the picture, with the real full names and associated traceability pragmas as follows:

```
Boot   : Boot_0_0       %go: 0%
Then1  : Then_1_0       %then: 1%
Else1  : Else_1_0       %else: 1% %go: 2%
Then3  : Then_3_0       %then: 3% %go 5% %emit: X%
Else3  : Else_3_0       %else: 3%
Go4    : Go_4_0         %go: 4%
R4     : PauseReg_4_0   %pause: 4%
Cont4  : Cont_4_0       %go: 3%
```

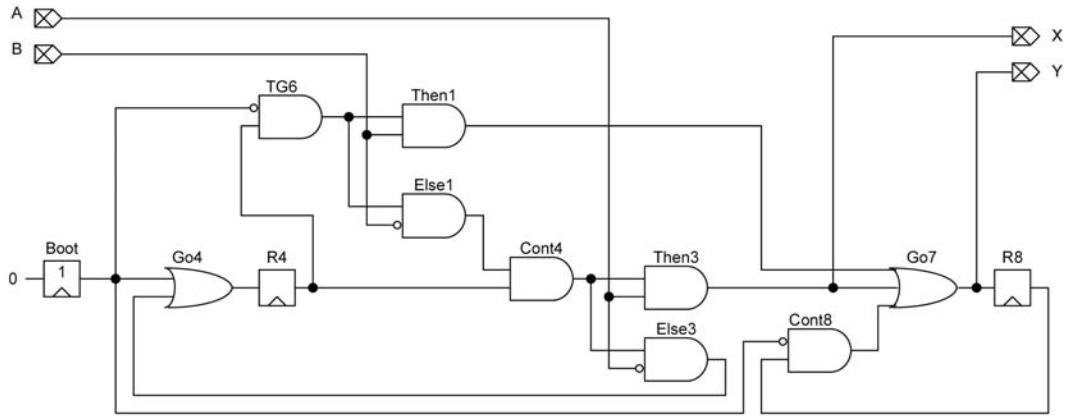


Figure 2: Esterel generated circuit

```
TG6    : ToGuard_6_0    %go: 1% %go: 6%
Go7    : Go_7_0         %go: 7% %go: 8% %emit: Y%
R4     : PauseReg_4_0   %pause: 8%
Cont8  : Cont_8_0
```

Long names are useful since they are printed in HDL or C and thus propagated down the synthesis chain. The name tells which function a gate is performing and where it comes from. The first number it contains is the index of the statement which generated the gate, while the second number is the module index that identifies to which instance of which submodule the statement belongs, see Section 2.4.

Pragmas add more traceability information to highlight source code at simulation time, to report error messages, and to perform ECOs. Notice that a gate can bear several pragmas. For instance, the `Then3` gate bears *`%then: 3%`*, which tells that the gate is 1 if the test statement 3 succeeds, *`%go: 5%`*, which tells that the same gate starts the statement 5 right away, and *`%emit: X%`*, which tells that `X` is emitted when the gate is 1. The last two pragmas are actually what remains of a gate `Go_5_0` that was initially created from `Emit` statement 5 but swept away since it simply acted as a buffer between the statements 3 and 7. Buffer sweeping carefully preserves pragmas.

Each `Pause` assembly statement generates a register, all the other statements generating combinational gates. The circuit works as follows:

- The initial tick is driven by the boot register, which has initial value 1 and input 0. The boot register loads the `R4` register generated by "`await A`".
- The wire out of `R4` is used for selection / resumption. It is anded with the negation of the boot register, which means global resumption. This triggers a test for `B`. If the test is true, control is transmitted to the `Go7` gate to emit `Y` and to `R8` pause on the `sustain` statement. If the test is false, control goes to the "`present A`" test 3 through the `Cont4` pause resumption gate, to test for `A`. If `A` is present, the `Then3` gate is set, `X` is emitted, the `Go7` gate is set, `Y` is emitted, and the sustain register `R8` is set. If `A` is absent, the `R4` await register is reloaded through `Go4`.

Generally speaking, the register part comes either from Esterel temporal statements such as `pause`, "`await S`", "`every S do ... end`", `loop ... each S`, which define sequential behavior, or from access to previous signal status `pre(S)` used for instance in the rising edge detection sequential expression "`S and not pre(S)`". The combinational logic is generated by the control and signal propagation statements such as signal emission "`emit S`", signal presence test "`if S then ... else ... end`", sequencing '`;`', parallel '`||`', etc.

2.3 User-defined names of registers

Generated register names are particularly important since they are fundamental for ECOs and usually preserved by the backend circuit synthesis flow. To make names more readable, one can associate a tag with a register-generating statement in the source Esterel code:

```
await@Wait A;
sustain@Sust Y
```

When setting a special Esterel compiler option `-eco`, the compiler embeds the tag in the register names, which become `PauseReg_Wait_4_0` and `PauseReg_Sust_8_0` instead of `PauseReg_4_0` and `PauseReg_8_0`. Furthermore, with this option, the register input gates are not subject to sweeping; they are named `PauseRegIn_Wait_4_0` and `PauseRegIn_Sust_8_0`. Since they are outputs of the combinational part, they are not swept by synthesis backends and easier to find when ECOs are needed.

2.4 Modular compilation and traceability

Esterel programs can involve calls to submodules. For instance, a Fifo is composed of a controller and of a memory linked by auxiliary signals:

```
module Fifo :
// import the fifo interface description
extends FifoInterface;
// declare local signals and instantiate components
signal {Read, Write} : Address in
   run FifoController
||
   run Memory
end signal
```

There are two main modes of HDL generation: *global*, where submodule assembly codes are inlined in the global module code; *modular*, in which submodules are compiled separately into HDL designs and instantiated in the main module HDL. Modular compiling is necessary for large applications, while global compiling makes it possible to deeply optimize modules by analyzing their global behavior. Mixed modes where some modules are inlined and others separately compiled is also available.

A potential problem with global compiling is gate name instability. To explain it, consider the global `Fifo` assembly code:

```
0:  0 Root: (1)
1:  0 SigScope: [Read, Write] (2, 20)
2:  1 Pause: (3) <0>                -- start of FifoController
3:  1 ...
```

```
...
20: 2 SigScope: [MemoryArray] (21) -- start of Memory
21: 2 If: [Write] (22, 35)
22: 2 ...
...
```

The module index tells which module a statement comes from: 0 for the main module, 1 for `FifoController`, 2 for `Memory`. In the circuit translation, submodule numbering is kept in gate names, as for `Then_21_2` generated by statement 21. However, the statement number itself is unstable: adding one statement by a local change in `FifoController` modifies the numbering of the statements and gates in the instantiated `Memory`. Using the `-eco` option of the Esterel compiler, we alleviate this problem by naming the gates relatively to their position in the submodule instance. In this case, the `Then_21_2` gate is renamed `Then_1_2`.

Finally, gates must be sorted according to control and data dependencies before being printed in HDL or C. Little changes in source code can have dramatic effect on the resulting order. To improve code stability for ECOs, we dissociate the gate definitions and the gate instantiation ordering, printing the gate definition in statement order and their instantiation in causal order. This will not be detailed further.

2.5 Traceability from graphical state machines

Esterel supports program design by hierarchical and concurrent safe state machines (SSMs), an evolution of C. André's SyncCharts [1] supporting Mealy and Moore machines. SSMs are translated into textual Esterel source code to be compiled. In SSMs, only terminal states without contents generate control registers; transitions, concurrency, and hierarchical macro-states only generate combinational logic. Terminal states can be named, and the names are propagated to generated Esterel `pause` statements using the @ tag symbol. Then, when using the `-eco` compiler option discussed in Section 2.4, graphical state names are pushed into HDL register names.

3 Sequential optimization

The reader may have found that the circuit in Figure 2 is too heavy for the purpose. This is typical for a circuit generated in syntax-directed translation by high-level synthesis, to which two kinds of optimizations must be applied:

- *Combinational optimizations*, which are classical in synthesis backends. They aggregate gates, remove logical redundancies, and simplify and reshape the logic using for instance don't care calculations [12]. In Figure 2, it is clear that the connection from `R4` to `Cont4` is redundant and can be removed. In practice, it is easier to leave such a simplification to the general combinational optimization pass performed by HDL synthesis front-ends.
- *Sequential optimizations*, which changes how the state is encoded by registers. This step is not done by standard synthesis backends and is found only in few systems such as Esterel or PBS [9]. We detail it in this section.

For sequential optimization, the simplest idea would be to count the number of reachable states of the design, say N, and to allocate $log_2(N)$ registers to hold them. In practice, this fails since the necessary state encoding / decoding circuitry tends to blow up. Finding the right register

allocation is difficult even for small designs. Furthermore, even if an optimal allocation is found, the obtained circuit can be quite bad in overall terms because of encoding combinational logic complexity. In practice, it is better to look for less aggressive register reduction schemes that try to ensure a better register / logic compromise by respecting the behavioral structure of the design,

Therefore, for Esterel optimization, we try to respect the initial encoding while removing its redundancies in a controlled way. We use algorithms presented in [13, 10, 11]. Here, we briefly present the three main ones: redundant register elimination, incompatible registers folding, and boot register elimination. We apply these techniques only on the control path of the circuit, because the data path needs to be handled very differently.

3.1 Redundant register elimination

We say that a register is *redundant* if it can be replaced by a function of the other registers without altering the sequential behavior of the circuit. For instance, in the circuit of Figure 2, we can replace `R8` by "`not(Boot or R4)`". The newly generated function can be merged with the rest of the combinational logic and simplified with it.

In [4], Madre and Coudert have presented a simple algorithm to check whether a given register r is redundant. Let $\vec{r}$ be the vector of all registers, and let $\mathcal{R}(\vec{r})$ be the characteristic function of the reachable state space of the circuit. Call $\mathcal{R}_r$ and $\mathcal{R}_{\overline{r}}$ the positive and negative cofactors of $\mathcal{R}$ by r, characterized by the Shannon decomposition $\mathcal{R} = r\mathcal{R}_r + \overline{r}\mathcal{R}_{\overline{r}}$. The cofactors are functions of the variables in $\vec{r}' = \vec{r} - \{r\}$. Then r is redundant if and only if the conjunction $\mathcal{R}_r(\vec{r}')\mathcal{R}_{\overline{r}}(\vec{r}')$ of the cofactors is 0. One can replace r either by the positive cofactor $\mathcal{R}_r(\vec{r}')$ or by the negation of the negative cofactor $\overline{\mathcal{R}_{\overline{r}}(\vec{r}')}$, whichever expression is simpler (they are functionally equal).

Of course, deciding whether a redundant register replacement is useful is a very difficult global optimization problem. We use heuristics based on the size of the cofactor supports (active variables).

3.2 Merging exclusive register groups

Consider the following Esterel program:

```
{ await A || await B };
{ await C || await D };
emit X
```

The behavior goes in two phases: waiting for the last of `A` and `B`, and waiting for the last of `C` and `D` to emit `X`. Call `RA`, `RB`, `RC`, and `RD` the 4 registers generated by the 4 `await` statements. Then `RA` and `RB` are concurrent and can take any of the possible 4 value pairs, and so are `RC` and `RD`. However, the `RA-RB` and `RC-RD` groups are dependent: the disjunction predicates $\texttt{RA} \vee \texttt{RB}$ and $\texttt{RC} \vee \texttt{RD}$ are exclusive, i.e. cannot be true together. Therefore, we can safely superpose the register pairs using an auxiliary switch register `SW`. For instance, we can use two registers `RE` and `RF` and set $\texttt{RA} = \texttt{RE} \wedge \texttt{SW}$, $\texttt{RB} = \texttt{RF} \wedge \texttt{SW}$, $\texttt{RC} = \texttt{RE} \wedge \overline{\texttt{SW}}$, and $\texttt{RD} = \texttt{RF} \wedge \overline{\texttt{SW}}$. We are left with 3 registers instead of 4 at the cost of introducing some combinational gates that could possibly be optimized away by combinational optimization. The effect would be more pronounced if the sequential components had more registers.

Notice that exclusive register folding can be detected from source code only, without computing reachable states. This is a clear advantage of explicit parallel / sequence temporal structures

over the classical HDL division between combinational and sequential processes. Knowing when to effectively apply exclusive register folding is subject to heuristics that are outside the scope of this paper.

3.3 Boot register elimination

In the circuit translation from Esterel, the initial instant is always triggered by the single `Boot` register, all other registers being initialized to 0. It may happen that this initial global state can be suppressed by removing the `Boot` register and changing the initial value of the other registers. In this case, the new circuit is obviously simpler. This is possible if there exists an initial value allocation to the other registers that leads to the same behavior and the same target states as the initial boot state. Algorithms are presented in

3.4 Reversible and traceable sequential optimization

An important property of the sequential optimizations we have presented is that they are *reversible*: the suppressed registers can be reconstructed from the kept ones. This will be very important for ECOs. In practice, we iteratively chain several optimization algorithms. To be able to reconstruct the old registers in function of the new ones, we could keep the undo information at every optimization step, which would be fairly error-prone. We find it much simpler to use a simpler trick: add the original registers as outputs of the circuit before running the optimization algorithms. The trick has an additional advantage: it makes it possible to use any optimizer and to optimize the circuit as a black box.

Technically speaking, call $\mathcal{C}$ the original circuit, $\vec{i}$ its input vector, $\vec{o}$ its output vector, and $\vec{r}$ its register vector. The circuit is determined by the combinational function $(\vec{o}, \vec{r}') = \mathcal{C}(\vec{i}, \vec{r})$, where $\vec{r}'$ is the new state vector for registers. Keeping the original registers as outputs amounts to consider a circuit $\mathcal{C}_{\vec{r}}$ of the form $(\vec{o}, \vec{r}', \vec{r}) = \mathcal{C}_{\vec{r}}(\vec{i}, \vec{r})$, where the input-output transformation from $\vec{r}$ to $\vec{r}$ is simply the identity function. Sequentially optimizing $\mathcal{C}_{\vec{r}}$ yields a new set of registers $\vec{s}$ and a new circuit $\mathcal{D}$ having type $(\vec{o}, \vec{s}', \vec{r}) = \mathcal{D}(\vec{i}, \vec{s})$. When running $\mathcal{D}$, on an input sequence, the values of the original register vector $\vec{r}$ of $\mathcal{C}$ is directly recovered from $\mathcal{D}$'s outputs.

From $\mathcal{D}$, we can recover two circuits: the one for actual synthesis, obtained by dropping the $\vec{r}$ outputs and performing further combinational optimization to remove combinational logic necessary only for the $\vec{r}$ fake output; the one for register reconstruction, obtained by retaining only the $\vec{r}$ output of $\mathcal{C}$, which we call $\mathcal{D}_{\vec{r}}$. The latter circuit summarizes all the undo parts of sequential optimizations, since any original register in $\vec{r}$ can be reconstructed by combinational logic from the final registers in $\vec{s}$ and the inputs in $\vec{i}$, which are the same as for $\mathcal{C}$.

Since we keep the $\vec{r}$ output, the sequential optimization of $\mathcal{D}$ may be less efficient than that of $\mathcal{C}$. In particular, state graph minimization cannot be used since it would collapse states.

For instance, here are the figures for a relatively large protocol state machine. The original circuit has 55 latches and yields a cell area of 959 when compiled with design compiler. The Esterel standard sequential optimization (heavy mode) yields a circuit with 34 latches and cell area 614. Keeping the original registers as output yields a circuit with 38 registers and area 652, which is 6% bigger.

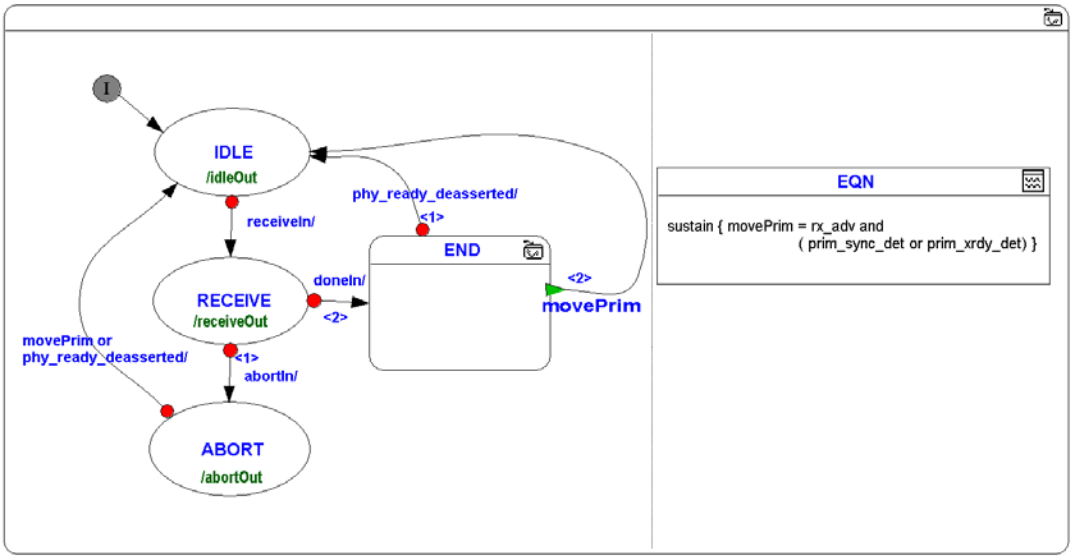


Figure 3: Adding a term in an equation

4 ECO examples in unoptimized mode

We present in this section different ECO manipulations corresponding to common bug fixes in a control path. The assumption is that the fix is easily done in the source model. But the circuit being already signed-off, it is not possible to use the modified model and generate a new layout. Instead, one must identify which changes of the already generated circuit correspond to the change in the Esterel model.

We consider here unoptimized designs. Section 5 will show how the same manipulations can be performed on sequentially optimized designs.

4.1 Adding a term in an equation

Consider the program of Figure 3. It is made of an explicit automaton and of a textual equation in parallel. The `END` state is a macrostate whose contents will be presented in the next section. Here, the goal is to add the `prim_pmreq_det` literal to the definition of `MovePrim`, which has been done in Figure 3 but not yet in the current HDL code, signed-off before this change. In Esterel Studio, by clicking on the `sustain` statement, we print the list of HDL variables generated by the equation. Then, we read their equations in the HDL code, which are as follows:

```
Go_106_0_L0D0 := Cont_108_0_L0D0 or Boot_0_0
Or_1_107_0_L0D0 := Status_prim_sync_det_S7_0
                   or Status_prim_xrdy_det_S8_0
And_0_107_0_L0D0 := Status_rx_adv_S5_0 and Or_1_107_0
SigExpTrue_107_0_L0D0 := Go_106_0 and And_0_107_0
```

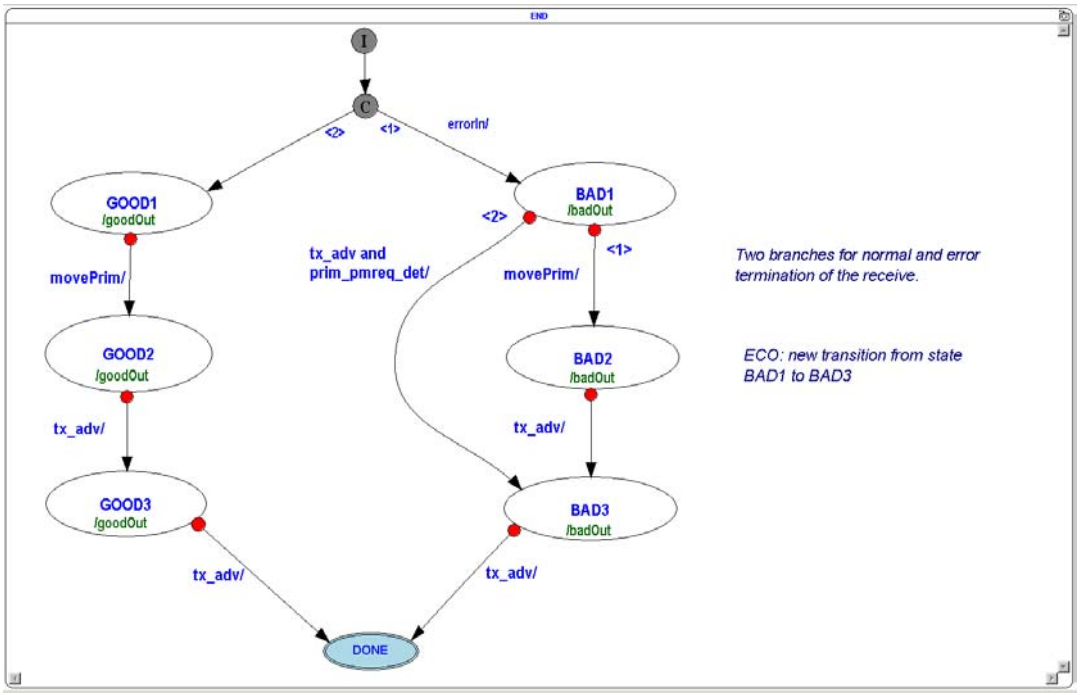


Figure 4: Adding a transition in a FSM

```
Status_movePrim_S15_0 := SigExpTrue_107_0
```

The signal status gates bear the names of the signals. We directly see that `movePrim` is emitted when `Go` is true and the decomposed Boolean expression (from the second and the third equations) is true. Notice that actual IO signals keep their names while local signals generate variables of the form `Status_movePrim`. The ECO simply consists in changing the second equation into

```
Or_1_107_0_L0D0 := Status_prim_sync_det_S7_0
                   or Status_prim_xrdy_det_S8_0
                   or prim_pmreq_det
```

4.2 Adding a transition between two states

Figure 4 shows the contents of the macro-state labelled `END` in Figure 3. We now want to add a transition between `BAD1` and `BAD3`, with trigger "`tx_adv and prim_pm_req_det`". The first step is to find the state registers and their inputs in the HDL code. Here, we can directly look for the `BAD1` and `BAD2` names, and we find registers `PauseReg_BAD1_53_0` and `PauseReg_BAD3_37_0` with respective combinational inputs `PauseRegIn_BAD1_53_0` and `PauseRegIn_BAD3_37_0`. Clicking on the states of the FSM also show the corresponding registers.

We need to modify the input equations of BAD1 as emphasized below (new code in italic):

```
PauseRegIn_BAD1_53_0 :=
  Go_51_0_L0D0
  and not(PauseReg_BAD1_53_0              -- in BAD1
          and tx_adv and prim_pmreq_det); -- trigger true
```

The change ensures that Bad1 is exited when the trigger is true. For BAD3, the change is more intricate:

```
PauseRegIn_BAD3_37_0 :=
  Go_35_0
  or (PauseReg_BAD1_53_0 -- in BAD1
       and tx_adv and prim_pmreq_det -- trigger true
       and not phy_ready_deasserted -- go to IDLE state
       and not Status_movePrim_S15_0); -- go to BAD2 state
```

Notice that actual IO signals keep their names while local signals generate variables of the form Status_movePrim. The last two negated signals are necessary to ensure that the new transition has a lower priority than the existing ones from BAD1 to IDLE (preemption as seen on Figure 3), and to BAD2.

4.3 Adding a new state

Adding a new state essentially consists in adding a new pause register and new equations for the transitions, which is done similar to the example explained in the previous section. Therefore, it is not more difficult.

4.4 Validating the ECOs

To prove ECO correctness, we use the fsm_verify sequential equivalence checker already mentioned in Section 1. We compile the new Esterel program into blif using the Esterel compiler, and we compile the new HDL program into blif using a synthesis tool such as Design Compiler and a blif library. We prove sequential equivalence between both blif designs using the fsm_verify. This technique has proved efficient for relatively large control-dominated designs. Of course, any HDL sequential equivalence technique is usable as well. If the modifications do not alter the register set, a combinational equivalence is sufficient.

5 ECOs on optimized designs

Let us assume we want to perform the ECO of Section 4.2 after sequential optimization. The direct identification of registers and gates has become impossible. However, if we use ECO-friendly optimization, we know how to re-generate the old registers as function of the new ones. Here, the register correspondence from the 11 original registers to the 5 new ones is as follows:

Original register	Function of new registers
PauseReg_107_0	not eco_31 or not eco_23 or eco_1
Boot_0_0	not eco_1 and eco_23 and eco_31
PauseReg_BAD3_37_0	eco_1 and eco_13
PauseReg_BAD1_53_0	eco_1 and eco_23 and eco_27
PauseReg_BAD2_45_0	eco_1 and eco_23 and eco_31
PauseReg_GOOD3_67_0	not eco_1 and eco_23 and eco_27
PauseReg_GOOD1_83_0	eco_1 and not eco_23 and eco_27
PauseReg_GOOD2_75_0	eco_1 and not eco_23 and eco_31
PauseReg_IDLE_100_0	not eco_1 and not eco_23 and eco_31
PauseReg_ABORT_17_0	not eco_1 and not eco_23 and eco_27
PauseReg_RECEIVE_92_0	not eco_1 and eco_13

5.1 Adding extra logic to exit a state

To add the required transitions from BAD1, we must first modify the way in which we exit state BAD1.

```
variable in_BAD1 :  std_logic;
variable eco_exit_BAD1 :  std_logic;
in_BAD1 := Aux_1_eco_1 and Aux_3_eco_23 and Aux_4_eco_27;
eco_exit_BAD1 := in_BAD1 and tx_adv and prim_pmreq_det;
V7_Aux_1_eco_1_last <= Aux_73_eco_2 and not eco_exit_BAD1;
V7_Aux_3_eco_23_last <= Aux_76_eco_24 and not eco_exit_BAD1;
V7_Aux_4_eco_27_last <= Aux_79_eco_28 and not eco_exit_BAD1;
```

The equation defining in_BAD1 comes directly from the table above. The state BAD1 is encoded in the optimized circuit with the three registers eco_1, eco_23, eco_27 having the value 1. So we complete the input equations of these registers such that their values are false when the control exits the state BAD1, which is represented by the new signal eco_exit_BAD1

5.2 Adding extra logic to enter a state

New logic is also added to enter state BAD3: the optimized registers encoding BAD3 are or-ed with the new transition trigger. Here also, one must take care of transitions from BAD1 to IDLE (triggered by phy_ready_deasserted) and to BAD2 (triggered by movePrim), which have a highest priority. Since Phy_ready_deasserted is a primary input, it is still present in the optimized circuit. The internal signal movePrim has been optimized away in the optimized circuit, and we must reconstruct it. Looking at the table above, we see that BAD2 is encoded as (eco_1 and eco_23 and eco_31). Therefore the transition from BAD1 to BAD2 is taken when in_BAD1 and the inputs of eco_1, eco_23 and eco_31 are all 1. The HDL code is modified to reflect changes on the inputs of registers encoding BAD3, i.e. eco_1 and eco_13:

```
variable BAD1_to_BAD2 :  std_logic;
variable BAD1_to_IDLE : std_logic;
variable exit_BAD1 :  std_logic;
BAD1_to_BAD2 := in_BAD1 and Aux_73_eco_2 and Aux_76_eco_24 and Aux_83_eco_32;
BAD1_to_IDLE := in_BAD1 and phy_ready_deasserted;
exit_BAD1 := BAD1_to_BAD2 or BAD1_to_IDLE;
```

```
V7_Aux_1_eco_1_last <= (Aux_73_eco_2 and not eco_exit_BAD1)
                       or (eco_exit_BAD1 and not exit_BAD1);
V7_Aux_2_eco_13_last <= Aux_68_eco_14
                        (eco_exit_BAD1 and not exit_BAD1);
```

Further these new Boolean expressions must be optimized to reduce extra logic.

5.3 Adding a new state

Adding a new state in an optimized circuit can be performed by adding a new register, or by using a previously unused encoding of the existing registers. The first approach is simple and done as already explained in this paper. The second approach is more difficult because the existing transitions to and from the states encoded with the reused registers are impacted.

5.4 Validating the ECOs

The ECO is validated in the same way as for unoptimized circuits and as explained in Section 4.4. Note that the correspondence table between the original registers and the new ones is useful for equivalence checking between the two versions of the same circuit: the unoptimized one and the optimized one. Indeed, we can build a new circuit by composing the optimized circuit $\mathcal{D}$ with $\mathcal{D}_{\vec{r}}$, this new circuit can be formally compared to the original circuit using a combinational equivalence checking. This is much more efficient than sequential equivalence checking.

6 Conclusion

We have shown that ECO is tractable even in the case of heavy sequential optimization of control-dominated designs written in Esterel. The three key ideas are modular synthesis, traceability from source code to circuits and back, and reversible sequential optimization. In particular, we have shown that sequential optimization reversibility makes it possible to transform the sequential ECO problem into the more classsical combinational ECO problem by reconstructing the removed registers. After ECO is performed, sequential equivalence techniques are used to verify circuit changes w.r.t. source code changes. The ECO flow we have described is supported by the Esterel Studio tool suite.

Acknowledgements: we thank Marc Perreaut for his help during this work.

References

[1] C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *Proc. CESA'96, Lille, France*, July 1996.

[2] G. Berry. Esterel on hardware. *Philosophical Transactions Royal Society of London A*, 339:87–104, 1992.

[3] G. Berry. *The Constructive Semantics of Pure Esterel*. Draft book, available at http://www.esterel.org, version 3, July 1999.

[4] O . Coudert C. Berthet, J-C. Madre. New ideas on symbolic manipulation of finite state machines. In *Proc. ICCAD*, 1990.

[5] O. Coudert, J-C. Madre, and H. Touati. Tiger 1.0 user manual. Technical report, Digital Equipment Paris Research Lab, 1993.

[6] Shi-Yu Huang, Kwang-Ting Cheng, Kuang-Chien Chen, Forrest Brewer, and Chung-Yang Huang. Aquila: An equivalence checking system for large sequential designs. *IEEE Trans. Comput.*, 49(5):443–464, 2000.

[7] J.-H R. Jiang and R. K. Brayton. On the verification of sequential equivalence. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 22(6):686–697, 2003.

[8] S. Singh M. Sheeran and G. Stalmarck. Checking safety properties using induction and a sat-solver. In *Proc. Formal Methods in Computer Aided Design (FMCAD 2000), Springer LNCS*, 2000.

[9] Andrew Seawright and Wolfgang Meyer. Partitioning and optimizing controllers synthesized from hierarchical high-level descriptions. In *Proceedings of the 35th annual conference on Design automation conference*, pages 770–775. ACM Press, 1998.

[10] E. Sentovich, H. Toma, and G. Berry. Latch optimization in circuits generated from high-level descriptions. In *Proc. International Conf. on Computer-Aided Design (ICCAD)*, 1996.

[11] E. Sentovich, H. Toma, and G. Berry. Efficient latch optimization using exclusive sets. In *Proc. Digital Automation Conference (DAC)*, 1997.

[12] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, University of California at Berkeley, 1992. Memorandum No. UCB/ERL M92/41.

[13] H. Touati and G. Berry. Optimized controller synthesis using Esterel. In *Proc. International Workshop on Logic Synthesis IWLS'93, Lake Tahoe*, 1993.

[14] C.A.J. van Eijk. Sequential equivalence checking based on structural similarities. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 19(7):814–819, 2000.

Structure-Driven Equivalence Verification for Circuits Optimized by Retiming and Combinational Synthesis

Maher Mneimneh, Karem Sakallah
{maherm,karem}@umich.edu
Electrical Engineering and Computer Science
University of Michigan

Sequential optimization techniques fall in two broad categories: state-based and structure-based [4]. Structure-based techniques optimize a circuit netlist by interleaving retiming and combinational synthesis in different ways to improve design metrics such as delay, area, and power. Examples of such optimizations include among others: peripheral retiming [7], architectural retiming [6], and iterative retiming and resynthesis [2]. While structure-based synthesis algorithms can handle relatively large netlists, current sequential equivalence verification algorithms are not as scalable and as robust. We believe that the absence of a reliable verification framework for such sequential optimization hinders their industrial acceptance despite potential improvements in design quality.

Sequential equivalence verification can be reduced to the problem of model checking safety properties. However, such a reduction fails to recognize the inherent structural and functional relation between verified circuits; such a relation is a direct consequence of transformations applied during structure-based sequential synthesis. We believe that realizing this relation is key to making sequential equivalence verification tractable.

van Eijk [2] describes an algorithm that utilizes structural similarities between nodes in the circuits being verified. The algorithm finds equivalent nodes by partition-refinement until a fixed point is reached. Since the algorithm is incomplete (might generate false negative results), van Eijk suggests adding "retiming logic"; still, false negates are not completely eliminated. In [1], this approach is combined with induction [9] to achieve completeness. However, induction-based methods might require increasing the induction depth to a large value before reaching an inductive invariant.

In this paper, we present a robust and complete verification algorithm for one class of structure-based sequential synthesis: a sequence of retiming steps followed or preceded by a sequence of combinational synthesis steps. An example of an optimization in this class is: retiming for minimum area followed by combinational synthesis for minimum delay followed by mapping to a technology library. In addition, we present possible generalizations of the approach to handle peripheral retiming and other structure-based sequential synthesis techniques.

The intuition behind the algorithm is the notion of *retiming invariants* [8]: retiming induces a functional relation among the latches of the two circuits; we call such a relation the retiming invariant. By computing the retiming invariant or a stronger invariant (another invariant that implies it), equivalence can be easily proved/disproved. The incompleteness of van Eijk's algorithm is a consequence of missing some of the retiming-induced latch relations critical to proving equivalence. We show that by extending partition refinement to include nodes at different time frames, the approach becomes complete. This extension generalizes the notion of "retimed logic" provided by van Eijk. We show how the equivalences at different time frames are utilized to construct retiming-induced functional relations. At the heart of this construction is an algorithm for computing generalized dominators [ref] in graphs. We present efficient methods to compute these dominators on directed acyclic graphs (the algorithms in [ref] work on control flow graphs). We also show the relation that exists between the number of time frames needed to prove/disprove equivalence and the number of latches moved across gates during retiming.

We implemented the above algorithm in C++. We present experimental results comparing this approach with van Eijk's method, induction and forward image computation for the ISCAS 89 circuits.

References

[1] P. Bjesse and K. Claessen, "SAT-based Verification Without State Space Traversal," in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, 2000.

[2] C. A. J. van Eijk, "Sequential Equivalence Checking Without State Space Traversal," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 1998.

[3] R. Gupta, "Generalized dominators and post-dominators," in *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, 1992.

[4] S. Hassoun, and T. Villa, "Optimization of Synchronous Circuits," in *Logic Synthesis and Verification*, Kluwer Academic Publishers, 2002.

[5] S. Hassoun and C. Ebeling, "Experiments in the iterative application of resynthesis and retiming," in The Proceedings of the International Workshop on *Timing Issues in the Specification and Synthesis of Digital Systems*, 1997.

[6] S. Hassoun and C. Ebeling, "Using precomputation in architecture and logic resynthesis", in *The Proceedings of the International Conference on Computer-Aided Design*, 1998.

[7] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovani-Vincentelli, "Retiming and resynthesis: Optimizing sequential networks with combinational techniques," IEEE Transaction on Computer-Aided Design, vol. 10, January 1991.

[8] M. Mneimneh, and Karem Sakallah, "REVERSE: Efficient sequential verification for retiming," in *Proceedings of the International Workshop on Logic and Synthesis*, 2003.

[9] M. Sheeran, S. Singh, and G. Stalmarck, "Checking safety properties using induction and a SAT-solver," in Formal Methods in System Design, 16(1), 2000.

For distribution in the participants' proceedings of the 5th International Workshop on Designing Correct Circuits, Barcelona, 27–28 March, 2004.

A Reflective Functional Language for Hardware Design and Theorem Proving*

Jim Grundy
Intel Corporation
Strategic CAD Labs, JF4-211
2111 NE 25th Avenue
Hillsboro, OR 97124, USA
jgrundy@ichips.intel.com

Tom Melham
Oxford University
Computing Laboratory
Wolfson Building, Parks Road
Oxford, OX1 3QD, England
melham@comlab.ox.ac.uk

John O'Leary
Intel Corporation
Strategic CAD Labs, JF4-211
2111 NE 25th Avenue
Hillsboro, OR 97124, USA
joleary@ichips.intel.com

ABSTRACT

This paper describes *reFLect*, a functional programming language with reflection features intended for applications in hardware design and verification. The *reFLect* language is strongly typed and similar to ML, but has quotation and antiquotation constructs. These may be used to construct and decompose expressions in the *reFLect* language itself. The paper motivates and presents the syntax and type system of this language, which brings together a new combination of pattern-matching and reflection features targeted specifically at our application domain. It also gives an operational semantics based on a new use of contexts as expression constructors.

1. INTRODUCTION

In this paper we describe *reFLect*, a new programming language for applications in hardware design and verification. The *reFLect* language is strongly typed and similar to ML [13] but has quotation and antiquotation constructs. These are used to construct and decompose expressions in the *reFLect* language itself and provide a form of reflection, similar to that in LISP but in a typed setting. The design of *reFLect* draws on the experience of applying an earlier reflective functional language called *FL* [1] to large-scale verification problems at Intel [16, 17, 18].

Hardware designs are modeled as *reFLect* programs. As with similar work based on Haskell [6, 22] or LISP [15, 19], a key capability is simulation of hardware models by executing functional programs. In *reFLect*, however, we also wish to do various operations on the abstract syntax of models written in the language—for example circuit design transformations [30]. Moreover, we want the *reFLect* language to form the core of a typed higher-order logic for specifying and verifying hardware properties [10, 23], and simultaneously the implementation language of a theorem prover for this logic.

Formal reasoning about hardware is performed using the Forte tool [16], which was originally designed around *FL* but now uses *reFLect*. Forte includes a theorem prover of similar design to the HOL system [11]. In such systems the object language is embedded as a data-type in the meta-language. Representing object-language expressions as a data-type makes it straightforward to implement the various term analysis and transformation functions required by a theorem prover. But separating the object-language and meta-language causes duplication and inefficiency. Theorem provers like HOL, for example, include special code for efficient execution of object-language expressions [5].

In *reFLect* we have made the data-structure used by the underlying language implementation to represent syntax trees available as a data-type within the language itself. Functions on that data-structure, like evaluation, are also made available. Our aim was to retain all the term inspection and manipulation abilities of the conventional theorem prover approach while borrowing an efficient execution mechanism from the meta-language implementation.

The logic of HOL-like systems is constructed following the model of Church's formulation of simple type theory [7], in which higher-order logic is defined on top of the λ-calculus. Our theorem prover follows this approach and constructs a variant of higher-order logic on top of the *reFLect* language. The reduction rules for the language in this paper are among the inference rules in our higher-order logic.

The applications just described give *intensional analysis* a primary role in *reFLect*. The design of our language is therefore different from staged functional languages like MetaML [33] and Template Haskell [28], which are aimed more at program generation and the control and optimization of evaluation. The *reFLect* language also provides a native pattern matching mechanism designed to make it easy to analyze the structure of code (and logical formulas).

In the sections that follow, we describe *reFLect* by presenting extensions to the λ-calculus that implement its key features. We first discuss in a bit more detail how the features of the language support our intended applications. We then present the syntax and type system, and give an operational semantics of evaluation. Details of a scheme for compiling *reFLect* into into the λ-calculus can be found in our technical report [12], from which this paper has been derived. This compilation scheme forms the basis of the *reFLect* implementation used at Intel.

2. MOTIVATION AND EXAMPLES

The *reFLect* language augments λ-calculus with a form of quotation, written by enclosing an expression between '⟪'

*This paper is dedicated to the memory of our friend and colleague Rob T. Gerth, who provided many insightful and helpful comments on the material contained herein.

and '⟩⟩'. The denotation of a quoted expression is its own abstract syntax. There is also an antiquotation mechanism, written by prefixing an expression with '^', that escapes the effect of a quotation. Quotation and antiquotation may also be used for pattern matching.

These *reFLect* features meet three related demands of our intended applications in hardware modeling and theorem proving. First, antiquotation and pattern matching make it easy to write term manipulation functions—specifically, the kinds of term manipulation needed to implement a theorem prover, but also term manipulations that do circuit transformation. Second, the reflection features of *reFLect* allow us to mix evaluation and theorem proving. Finally, *reFLect* quotation provides a flexible framework in which to embed both logics and domain-specific languages.

2.1 Term Manipulation

Theorem proving systems like HOL have many ML functions for constructing and destructing terms the object language. Function applications are a typical example: HOL provides an ML function that maps terms 'f' and 'x' to the function application term '$f{\cdot}x$', and an ML function that takes an application term '$f{\cdot}x$' apart and returns the subterms 'f' and 'x'.

Analogous functions can be implemented in *reFLect* for constructing and destructing quoted *reFLect* applications. The definitions are as follows:

$$\mathsf{let}\ \mathit{make\_apply} = \lambda f.\,\lambda x.\,\langle\!\langle \hat{}f{\cdot}\hat{}x \rangle\!\rangle$$
$$\mathsf{let}\ \mathit{dest\_apply} = \lambda \langle\!\langle \hat{}f{\cdot}\hat{}x \rangle\!\rangle.\,(f, x)$$

A more complex example is the *reFLect* function below, which traverses a quoted *reFLect* expression and swaps the operands of any occurrence of the infix function '+'.

$$\begin{aligned}\mathsf{letrec}\ \mathit{comm} = {} & \lambda\langle\!\langle \hat{}x + \hat{}y \rangle\!\rangle.\,\langle\!\langle \hat{}(\mathit{comm}{\cdot}y) + \hat{}(\mathit{comm}{\cdot}x) \rangle\!\rangle \\ \mid {} & \lambda\langle\!\langle \hat{}f{\cdot}\hat{}x \rangle\!\rangle.\,\langle\!\langle \hat{}(\mathit{comm}{\cdot}f){\cdot}\hat{}(\mathit{comm}{\cdot}x) \rangle\!\rangle \\ \mid {} & \lambda\langle\!\langle \lambda\hat{}p.\,\hat{}b \rangle\!\rangle.\,\langle\!\langle \lambda\hat{}p.\,\hat{}(\mathit{comm}{\cdot}b) \rangle\!\rangle \\ \mid {} & \lambda\langle\!\langle \lambda\hat{}p.\,\hat{}b \mid \hat{}a \rangle\!\rangle.\,\langle\!\langle \lambda\hat{}p.\,\hat{}(\mathit{comm}{\cdot}b) \mid \hat{}(\mathit{comm}{\cdot}a) \rangle\!\rangle \\ \mid {} & \lambda x.\,x\end{aligned}$$

For example, the application $\mathit{comm}{\cdot}\langle\!\langle \lambda x.\,m * x + c \rangle\!\rangle$ evaluates to $\langle\!\langle \lambda x.\,c + m * x \rangle\!\rangle$.

The merit of this pattern-matching style of programming is that it gives compact code that is also easy to read and understand. Aasa, Petersson and Synek [2] advocate a similar mechanism, called *quotation patterns*, for manipulating object-language expressions within a meta-language. Our experience of implementing a theorem prover in *reFLect* confirms the practical utility of this approach.

2.2 Reflection

The object logic of systems like HOL is typically a version of higher order logic defined on top of the λ-calculus. Logically, the construction follows the lines of Church's formulation of simple type theory [7], in which primitive symbols are added for certain constants such as equality and the quantifiers are defined using λ abstraction. The logic inherits a semantics for term equality from the λ-calculus; in particular, it inherits the various reduction rules of the λ-calculus, which appear in logic as inference rules.

Defining a logic on top of *reFLect* gives a higher order logic that includes the *reFLect* reduction rules in the same way. In a theorem prover implemented in *reFLect*, the data representations of both the object and meta-languages are the same. Hence reduction by execution of *reFLect* programs also directly implements reduction by formal inference in the logic. Theorem provers with a separate object and meta languages, on the other hand, need to include special code for efficient execution of object-language expressions [3, 5].

This link between program execution and logical inference provides a form of *reflection* [14] in the *reFLect* theorem prover. We can do equality proofs by term reduction in the theorem prover efficiently, just by evaluating the *reFLect* expressions. In particular, we can prove theorems by using the *reFLect* evaluation mechanism to evaluate the statement of the theorem to true. Conversely, we may obtain any *reFLect* program that evaluates to true as a theorem of our logic.

In a system like Forte, the invocation of a model checker is just a *reFLect* function call, so reflection also provides a logically principled connection between theorems in higher order logic and model checking results. A similar mechanism called *lifted-FL* [1] was available in earlier versions of Forte, but *reFLect* provides much richer possibilities. For example, one can use quantifiers to create a bookkeeping framework that cleanly separates logical content from model-checking control parameters.

The unification of object language and meta-language data representations also allows efficient evaluation to be incorporated into term rewriting in the theorem prover. If the *reFLect* evaluator supports evaluation of 'open code', then this can be extended to do efficient term simplification too.

2.2.1 Embeddings

The quotation construct in *reFLect* makes the whole of the *reFLect* language available as an embedded language within the *reFLect* programming language itself. Quotations are essentially a 'deep' embedding, in which the embedded language is represented as a data type. This is in contrast to a 'shallow' embedding, in which the embedded language is just a sublanguage of the language in which it is embedded. In a shallow embedding, one typically defines a collection of functions to represent various components of the language. The embedded language itself is then just the collection of all programs that can be written using these functions.

The merit of a shallow embedding is that it directly inherits an efficient execution mechanism—as well as an evaluation semantics—from the programming language in which it is defined. On the other hand, one cannot define functions that inspect the syntactic form of phrases in the embedded language. For example one cannot define functions that do transformations of expressions in the embedded language. For this, the language must be deeply embedded as a data type.

In *reFLect* we can have much of the power of a deep embedding in a shallow embedding; since we have a built-in deep embedding of all of *reFLect*, we also have a deep embedding of any sublanguage of it. Suppose, for example, we define a shallow embedding of an HDL into a functional programming language. We could then express circuits in the HDL as functional programs and simulate them by execution. On the other hand, if we define the HDL as a data type within a functional language, then we can write code that inspects and transforms HDL programs, but we cannot evaluate them

$$
\begin{array}{rcll}
\sigma, \tau, \ldots & ::= & \alpha \mid \beta \mid \gamma \mid \ldots & \text{– A type variable} \\
& \mid & (\sigma_1, \ldots \sigma_n)c & \text{– A compound type}
\end{array}
$$

Figure 1: The Syntax of Types

$$
\begin{array}{lcl}
\mathit{vars}\,\alpha & = & \{\alpha\} \\
\mathit{vars}(\sigma_1, \ldots \sigma_n)c & = & \mathit{vars}\,\sigma_1 \cup \ldots \mathit{vars}\,\sigma_n
\end{array}
$$

Figure 2: The Type Variables of a Type

without writing our own evaluator. In *reFLect*, however, we can do shallow embedding and get execution, but also use quotations and pattern matching to inspect and transform the phrases of the embedded HDL.

3. SYNTAX

The syntax of *reFLect* is similar to that of the typed λ-calculus, but with function abstraction constructed over general patterns, rather than just variables, and with primitive syntax for quotations and anti-quotations.

3.1 Types

The *reFLect* language is simply typed in the Hindley-Milner style, like ML. A type may be a type variable, written with a lower-case letter from the start of the Greek alphabet: α, β, etc.; or a compound type, made up of a type operator applied to a list of argument types. We use lower-case letters from the end of the Greek alphabet, σ, τ, etc., for syntactic meta-variables ranging over types. Type operators are usually written post-fix, but certain binary type operators, such as $\rightarrow$ and $\times$, are written infix. Atomic types, like *int* and *bool*, are considered to be zero-ary type operators applied to empty lists of arguments. The *reFLect* type system contains one interesting atomic type: *term*, the type of a quoted *reFLect* expression. Figure 1 shows the syntax of the *reFLect* type system assuming a syntactic class of type operator symbols, written c.

We assume the existence of a meta-linguistic function *vars* from types to the sets of type variables that occur in them. We also apply *vars* to sets of types, implicitly taking the union of their sets of variables. Figure 2 defines the function *vars*.

3.1.1 Type Instantiation

A *type instantiation* is a mapping from type variables to types that is the identity on all but finitely many arguments. We use the meta-variables ϕ and χ to stand for type instantiations. We will write $\mathit{dom}\,\phi$ for the *domain* of ϕ, meaning the set of variables for which ϕ is not the identity. If $\mathit{dom}\,\phi = \{\alpha_1, \ldots \alpha_n\}$ and $\phi\,\alpha_i = \sigma_i$ for $1 \leq i \leq n$, then we sometimes write ϕ as $[\sigma_1, \ldots \sigma_n/\alpha_1, \ldots \alpha_n]$.

Every type instantiation induces a map from types to types. For any type σ and instantiation ϕ we will write σ_ϕ for the result of applying the map induced by ϕ to σ. The induced map is described in Figure 3.

$$
\begin{array}{lcl}
\alpha_\phi & = & \phi\,\alpha \\
(\sigma_1, \ldots \sigma_n)c_\phi & = & (\sigma_{1\phi}, \ldots \sigma_{n\phi})c
\end{array}
$$

Figure 3: Type Instantiation

$$
\begin{array}{rcll}
\Lambda, \mathrm{M}, \ldots & ::= & k^\circ_\circ\,\sigma & \text{– Constant} \\
& \mid & v^\circ_\circ\,\sigma & \text{– Variable} \\
& \mid & \lambda\Lambda.\,\mathrm{M} & \text{– Abstraction} \\
& \mid & \lambda\Lambda.\,\mathrm{M} \mid \mathrm{N} & \text{– Alternation} \\
& \mid & \Lambda{\cdot}\mathrm{M} & \text{– Application} \\
& \mid & \langle\!\langle \Lambda \rangle\!\rangle & \text{– Quotation} \\
& \mid & \hat{}\Lambda^\circ_\circ\,\sigma & \text{– Antiquotation}
\end{array}
$$

Figure 4: The Syntax of Expressions

3.2 Expressions

The syntax of *reFLect* expressions, shown in Figure 4, is an extension of the syntax of the λ-calculus. Uppercase letters from the middle of the Greek alphabet, Λ, M, etc., range over expressions. We assume the existence of syntactic classes of constant names and variable names, ranged over by k and v respectively. For clarity of presentation, we will write constants such as $+$, $*$, $\vee$ and , (pairing) in infix position. The syntax requires explicit type annotations for constants, variables, and antiquotations. For example, $v^\circ_\circ\,\sigma$ is a variable with name v and type σ. We may omit type annotations when the type is easily inferred.

Several extensions over the simple λ-calculus are apparent from the grammar. These are discussed below.

3.2.1 Constants

Constants are not theoretically necessary in a presentation of the λ-calculus and are therefore often omitted. They are, however, important in any practical logic and so we include them here. Constants also play a special role in *reFLect* by facilitating a restricted form of polymorphism.

A practical functional language would normally be based on an extension of the λ-calculus with a polymorphic local let construct. However, simple type-theories based on such extended λ-calculi exhibit Girard's Paradox [8]. Since we use the *reFLect* language as the foundation for such a logic we must eschew this extension. This does not mean that our language lacks all polymorphism; constants may be polymorphic. We assume a top level let command that defines a constant to stand for a closed, possibly polymorphic, expression. The type checking rules given later in section 4.2 will allow that constant to be used at any instance of its polymorphic type. The logical soundness of this restricted form of polymorphism is exhibited in Pitts's semantics for the HOL logic [11].

3.2.2 Quotations

A *reFLect* expression may contain a quoted *reFLect* expression. These are written using the form $\langle\!\langle \Lambda \rangle\!\rangle$. Note that $1 + 2$ is considered semantically equal to 3, but that $\langle\!\langle 1 + 2 \rangle\!\rangle$ is considered semantically different from $\langle\!\langle 3 \rangle\!\rangle$. The expressions $1 + 2$ and 3 both denote the same integer value, namely 3. The expression $\langle\!\langle 1 + 2 \rangle\!\rangle$ denotes the abstract syntax tree of the expression $1 + 2$, which is different from the abstract syntax tree of the expression 3.

The *reFLect* language also has an antiquotation operation, which is used to remove the quotes from around its argument. Antiquotation is written $\hat{}\Lambda^\circ_\circ\,\sigma$ or (omitting the type) just $\hat{}\Lambda$ and may be used only inside quotation. Section 6.1

will explain how in certain circumstances subexpressions of the form $\hat{}\langle\!\langle \Lambda \rangle\!\rangle$ may be reduced to Λ. As an example, consider the expression $\langle\!\langle 1 + \hat{}\langle\!\langle 2 + 3 \rangle\!\rangle \rangle\!\rangle$. This expression may be reduced to to $\langle\!\langle 1 + (2 + 3) \rangle\!\rangle$. The expressions are considered semantically equal, denoting the same abstract syntax tree.

3.2.3 Abstractions

In the λ-calculus each abstraction binds a single variable. In *reFLect* an expression may appear in the binding position of an abstraction, which then binds all the free variables of that expression. Not all such expressions will be executable, though all are meaningful. We leave a precise description of which expressions are executable until later. Abstractions with a quotation in the binding position are evaluated by pattern matching. By using these facilities we may write an expression like $(\lambda\langle\!\langle \hat{}x + \hat{}y \rangle\!\rangle.\, \langle\!\langle \hat{}y + \hat{}x \rangle\!\rangle) \cdot \langle\!\langle 1 + 2 \rangle\!\rangle$, which is semantically equal to $\langle\!\langle 2 + 1 \rangle\!\rangle$.

Not all attempts to execute an application by pattern matching will succeed, so *reFLect* includes an alternation construct that can be used to try alternative patterns. Using this construct we may write the following function that commutes the arguments of quoted additions and multiplications:

$$\lambda\langle\!\langle \hat{}x + \hat{}y \rangle\!\rangle.\, \langle\!\langle \hat{}y + \hat{}x \rangle\!\rangle \mid \lambda\langle\!\langle \hat{}x * \hat{}y \rangle\!\rangle.\, \langle\!\langle \hat{}y * \hat{}x \rangle\!\rangle$$

Most logical languages omit pattern matching from their abstract syntax so as to simplify their semantics. We considered doing this with *reFLect*, but decided against it for two reasons. The first is that pattern matching quoted expressions seemed the most natural interface for inspecting and destructing term values. The second is that we wish to support reasoning about all well-founded *reFLect* functions including those that make use of pattern matching, which the implemented language also supports on algebraic data-types. The earlier *FL* system excluded pattern matching from the logical language, and so reasoning was performed on expressions after pattern matching had been translated into conditional expressions. In *reFLect* we support reasoning about expressions in a form closer to the surface syntax in which the user wrote them.

Syntactically, any *reFLect* expression can appear as a pattern. A natural alternative would be to have a separate syntactic class of patterns, but this was rejected because in the implemented language we allow a rather broad class of patterns. These include literal constants for integers, booleans and string, as well as an open-ended class of patterns built up from data-type constructors for free algebras. A separate grammar for patterns would therefore have to duplicate much of the expression language anyway. In addition, the expression of algorithms that traverse expressions would be more complicated, with separate cases for patterns and other expressions. Users often write expression-traversal code in theorem proving and design transformation applications—unlike in a compiler, where the developers write it once.

We could treat patterns as a subtype of expressions, and use a runtime check when an expression is antiquoted into a pattern position to confirm that it is a valid pattern. We may add such a check in a future version of *reFLect* if our experience suggests it is warranted and we can devise an implementation that does not degrade the performance of theorem proving algorithms that make heavy use of antiquotation for expression construction.

3.3 Contexts

For later use in describing the semantics of *reFLect*, we introduce the notation of a *context* to represent an expression with a number of *holes* that occur at specific subexpression positions in the abstract syntax tree. The notion of context we use here is similar to that readers may be familiar with from other language descriptions, except that the holes in our contexts are typed.

Formally, contexts are described by the same grammar as expressions, with the addition of a new production to represent a hole.

$$\begin{array}{lcll} \Lambda, \mathrm{M}, \ldots & ::= & \ldots & \textit{(as in Figure 4)} \\ & \mid & \_ \colon \sigma & -\text{ A hole} \end{array}$$

A hole is represented by the symbol '_' annotated by a type. We may omit type annotations on holes in a context when they are irrelevant or easily inferred.

We use the calligraphic letters, $\mathcal{C}$, $\mathcal{D}$, etc., as syntactic meta-variables ranging over contexts. We will use the notation $\mathcal{C}[\_ \colon \sigma_1, \ldots \_ \colon \sigma_n]$ to indicate that the context $\mathcal{C}$ has the n holes shown. The order in which the holes are indicated is unimportant, except that it be must fixed for any given context. We write $\mathcal{C}[\Lambda_1, \ldots \Lambda_n]$ to stand for the expression resulting from a context $\mathcal{C}[\_ \colon \sigma_1, \ldots \_ \colon \sigma_n]$, where σ_1, ... σ_n are the types of Λ_1, ... Λ_n respectively, in which each hole $\_ \colon \sigma_i$ has been filled by expression Λ_i. Note that this is different from the usual notion of expression *substitution*, in that there is no renaming to avoid variable capture.

4. STATIC SEMANTICS

In this section we introduce the two well-formedness criteria for expressions. The first is a notion of 'level' which constrains the nesting of quotations and antiquotations allowed in an expression. The second is a notion of strong typing.

4.1 Level

We use the term *level* to mean the number of quotations that surround a subexpression. The level of a quoted subexpression is one higher than the level of the surrounding expression. The level of an antiquoted subexpression is one lower than the level of the surrounding subexpression. The level of an entire expression is zero, and no expression may occur at negative level.

Level is an important notion in *reFLect* because it affects variable binding and reduction. Generally speaking, expressions that occur at level zero may be reduced while those that occur at a higher level may not. For example the normal form of the expression $(1 + 2, \langle\!\langle 1 + 2 \rangle\!\rangle)$ is $(3, \langle\!\langle 1 + 2 \rangle\!\rangle)$ because the first occurrence of $1 + 2$ occurs at level zero in the expression and therefore may be reduced, while the second occurrence is at level one and therefore may not.

We formalize our notion of level in relation to contexts. Since all expressions may be considered as contexts with no holes, the definitions and properties we describe for contexts also apply to expressions. We consider a context to be well formed only if all its holes occur at level zero and no portion of the context occurs at a negative level. We will say that such a context is *level consistent*. For example, $\hat{}\_ + 1$ not level consistent, but $\langle\!\langle \hat{}\_ + 1 \rangle\!\rangle$ is.

Figure 5 formalizes our notion of a level consistent context by defining judgments of the form $n \vdash \mathcal{C}$, which should be read

$$\frac{}{0 \vdash \_ \circ \sigma} \qquad \frac{}{n \vdash k \circ \sigma} \qquad \frac{}{n \vdash v \circ \sigma}$$

$$\frac{n \vdash \mathcal{C} \quad n \vdash \mathcal{D}}{n \vdash \lambda\mathcal{C}.\,\mathcal{D}} \qquad \frac{n \vdash \mathcal{C} \quad n \vdash \mathcal{D} \quad n \vdash \mathcal{E}}{n \vdash \lambda\mathcal{C}.\,\mathcal{D} \mid \mathcal{E}}$$

$$\frac{n \vdash \mathcal{C} \quad n \vdash \mathcal{D}}{n \vdash \mathcal{C}{\cdot}\mathcal{D}} \qquad \frac{n+1 \vdash \mathcal{C}}{n \vdash \langle\!\langle \mathcal{C} \rangle\!\rangle} \qquad \frac{n \vdash \mathcal{C}}{n+1 \vdash (\hat{\ } \mathcal{C} \circ \sigma)}$$

Figure 5: A Level Consistent Context, $n \geq 0$

as '$\mathcal{C}$ is level consistent at level n'. We may read judgments of the form $0 \vdash \mathcal{C}$ as simply '$\mathcal{C}$ is level consistent'. If the unique derivation of $0 \vdash \mathcal{C}$ contains a subderivation with the intermediate conclusion $n \vdash \mathcal{D}$, then we say that '$\mathcal{D}$ occurs at level n in $\mathcal{C}$'.

The following properties follow from the definition of level consistency.

PROPOSITION 1. *If $\mathcal{C}$ contains no holes and $n \vdash \mathcal{C}$, then $m \vdash \mathcal{C}$ for any $m \geq n$.*

PROPOSITION 2. *For any n and $\mathcal{C}$, there is at most one derivation concluding $n \vdash \mathcal{C}$.*

PROPOSITION 3. *If $\mathcal{C}$ contains one or more holes, then there exists at most one n such that $n \vdash \mathcal{C}$.*

PROPOSITION 4. *If Λ is an expression such that $1 \vdash \Lambda$ then there is a unique context $\mathcal{C}[\_ \circ \sigma_1, \ldots \_ \circ \sigma_n]$ and set of expressions $\mathrm{M}_1, \ldots \mathrm{M}_n$ such that $\mathcal{C}[\hat{\ }\mathrm{M}_1 \circ \sigma_1, \ldots \hat{\ }\mathrm{M}_n \circ \sigma_n]$ is syntactically identical to Λ and $0 \vdash \mathcal{C}[\_ \circ \sigma_1, \ldots \_ \circ \sigma_n]$.*

Proposition 4 allows us to treat contexts as a form of general constructor for quoted expressions. We will use an expression of the form $\langle\!\langle \mathcal{C}[\hat{\ }\Lambda_1 \circ \sigma_1, \ldots \hat{\ }\Lambda_n \circ \sigma_n] \rangle\!\rangle$ under the condition $0 \vdash \mathcal{C}[\_ \circ \sigma_1, \ldots \_\sigma_n]$ to stand for any quoted expression with level zero subexpressions $\Lambda_1, \ldots \Lambda_n$. Many of the remaining figures contain recursive definitions over the structure of expressions that use this property to give the case for quoted expressions. Figures 6 and 9 are typical examples. This mechanism allows us to write our structural definitions such that they traverse only the level zero portions of an expression. This contrasts with the presentation technique used for other reflective languages [33, 28] in which the entire term in traversed, and the traversal function tracks the level of the current expression.

4.2 Typing

All quoted expressions in *reFLect* have the same type, *term*. In *FL*, the type of a quoted expression depended on what was inside the quote [1]. For example, $\langle\!\langle x + y \rangle\!\rangle$ had type *int term*, while $\langle\!\langle p \vee q \rangle\!\rangle$ had type *bool term*. The idea was similar to the code type `<σ>` of MetaML [33]. But this scheme means that certain functions that destruct or traverse the structure of an expression cannot be typed. Such functions are common in our target application domain of theorem proving; the functions in section 2 are typical examples.

$$\frac{(\mathit{mgtype}\, k)_\phi = \sigma}{\vdash k \circ \sigma : \sigma} \qquad \frac{}{\vdash v \circ \sigma : \sigma} \qquad \frac{\vdash \Lambda : \sigma \quad \vdash \mathrm{M} : \tau}{\vdash \lambda\Lambda.\,\mathrm{M} : \sigma \to \tau}$$

$$\frac{\vdash \Lambda : \sigma \quad \vdash \mathrm{M} : \tau \quad \vdash \mathrm{N} : \sigma \to \tau}{\vdash \lambda\Lambda.\,\mathrm{M} \mid \mathrm{N} : \sigma \to \tau} \qquad \frac{\vdash \Lambda : \sigma \to \tau \quad \vdash \mathrm{M} : \sigma}{\vdash \Lambda{\cdot}\mathrm{M} : \tau}$$

$$\frac{\begin{array}{l} 0 \vdash \mathcal{C}[\_ \circ \sigma_1, \ldots \_ \circ \sigma_n] \\ \vdash \Lambda_1 : \mathit{term} \quad \ldots \quad \vdash \Lambda_n : \mathit{term} \\ \vdash \mathcal{C}[v_1 \circ \sigma_1, \ldots v_n \circ \sigma_n] : \tau \end{array}}{\vdash \langle\!\langle \mathcal{C}[\hat{\ }\Lambda_1 \circ \sigma_1, \ldots \hat{\ }\Lambda_n \circ \sigma_n] \rangle\!\rangle : \mathit{term}}$$

Figure 6: A Well Typed Expression

Pašalić et al. show how to use dependent types to address the problem of typing transformation routines [26]. But there are functions, such as finding free variables, that are important for implementing theorem provers and which still cannot be typed in a dependent type system. Even if it were possible to type such routines with dependent types we would reject this option because we wish to present our end-users, practicing hardware design engineers, with the simplest type system that meets their needs. By giving all quoted expressions the same type, *term*, we can type such expressions in a Hindley-Milner type system. The same decision is made for similar reasons in Template Haskell [28].

This means, of course, that in *reFLect* some type-checking must be done at run time.[1] For example the expression $\langle\!\langle 1 + \hat{\ }x \rangle\!\rangle$ is well-typed and requires x to be of type *term*. But the further requirement that x is bound only to integer-valued expressions cannot be checked statically; it must be enforced at run time.

This design decision goes against the common functional programming ideal of catching as many type errors as possible statically. Our approach, however, is similar to the way typing is handled in conventional theorem-proving systems that have a separate meta-language and object-language, such as HOL. Both languages are strongly typed, but evaluating a meta-language expression may attempt to construct an ill-typed object language expression, resulting in a run-time error. Our experience in the theorem proving domain is that this seemingly 'late' discovery of type errors is not a problem in practice.

4.2.1 A Well Typed Expression

We say Λ is well-typed with type σ if it is level consistent and we may derive the judgment $\vdash \Lambda : \sigma$ by the rules of Figure 6. Some of the rules merit explanation:

- We suppose that each constant symbol k has an associated *most general type*, $\mathit{mgtype}\, k$. The type of a constant named k may be any instance of this type.
- A variable may be explicitly annotated with any type, and it is well-typed with this type.
- If the body of a quotation is well-typed with some type σ then the quotation is well-typed with type *term*. The type of the body does not figure in the type of the quoted expression as a whole.

[1] Unlike Template Haskell, in which second-level type errors can still be caught at compile time.

$$\frac{(\mathit{mgtype}\, k)_\phi = \sigma}{\Gamma \vdash k \,\hat{\circ}\, \sigma : \sigma} \qquad \frac{(v \mapsto \sigma) \in \Gamma}{\Gamma \vdash v \,\hat{\circ}\, \sigma : \sigma}$$

$$\frac{\Delta \vdash \Lambda : \sigma \quad (\Delta \cup \Gamma) \vdash \mathrm{M} : \tau}{\Gamma \vdash \lambda\Lambda.\, \mathrm{M} : \sigma \to \tau}$$

$$\frac{\Delta \vdash \Lambda : \sigma \quad (\Delta \cup \Gamma) \vdash \mathrm{M} : \tau \quad \Gamma \vdash \mathrm{N} : \sigma \to \tau}{\Gamma \vdash \lambda\Lambda.\, \mathrm{M} \mid \mathrm{N} : \sigma \to \tau}$$

$$\frac{\Gamma \vdash \Lambda : \sigma \to \tau \quad \Gamma \vdash \mathrm{M} : \sigma}{\Gamma \vdash \Lambda \cdot \mathrm{M} : \tau}$$

$$\frac{\begin{array}{l} 0 \vdash \mathcal{C}[\_ \,\hat{\circ}\, \sigma_1, \ldots \_ \,\hat{\circ}\, \sigma_n] \\ \Gamma \vdash \Lambda_1 : \mathit{term} \quad \ldots \quad \Gamma \vdash \Lambda_n : \mathit{term} \\ \Delta \vdash \mathcal{C}[v_1 \,\hat{\circ}\, \sigma_1, \ldots v_n \,\hat{\circ}\, \sigma_n] : \tau \end{array}}{\Gamma \vdash \langle\!\langle \mathcal{C}[\hat{\ }\Lambda_1 \,\hat{\circ}\, \sigma_1, \ldots \hat{\ }\Lambda_n \,\hat{\circ}\, \sigma_n] \rangle\!\rangle : \mathit{term}}$$

Figure 7: Type Inference

- An antiquotation expression will be well typed if the body of the antiquotation has type *term*, regardless of the type annotated on the antiquote.

PROPOSITION 5. *For any Λ there is at most one type σ such that $\vdash \Lambda : \sigma$.*

4.2.2 Type Inference

Type inference in *reFLect* takes user input and constructs well-typed expressions, attaching the type annotations required to variables, constants and antiquotations. Users need not include these annotations in their input, though they may if they wish a more restricted type than would otherwise be inferred. The type inference algorithm used is essentially the Hindley-Milner algorithm, which performs type-checking relative to an environment associating each variable with its type. The algorithm is different for *reFLect* in that it performs type checking relative to the typing environment on the top of a stack of such environments. A fresh environment is pushed for each quotation. The stack is popped while traversing an antiquotation.

The *reFLect* type inference system will produce well-typed expressions with the most general type consistent with the rules of figure 7. Each judgment of the form $\Sigma \vdash \Lambda : \sigma$ should be interpreted as meaning that the expression Λ may have the type σ under the environment Σ. The environment of a judgment is a map, Γ, from variable names to their types.

PROPOSITION 6. *If $\Sigma \vdash \Lambda : \sigma$ can be deduced from the type inference rules in Figure 7, then $\vdash \Lambda : \sigma$ may be deduced from the type checking rules in Figure 6.*

4.2.3 Variables and Types

In *reFLect* the identity of a variable is determined by the combination of its name and type. A well-typed expression may have two or more (different) variables with the same name but different types. The type inference algorithm will never produce such an expression, but they may arise as a result of evaluation. For example, the expression $\langle\!\langle \hat{\ }\langle\!\langle x \,\hat{\circ}\, \alpha \to \beta \rangle\!\rangle \cdot \hat{\ }\langle\!\langle x \,\hat{\circ}\, \alpha \rangle\!\rangle \rangle\!\rangle$ may be reduced using the rules in section 6.1 to $\langle\!\langle x \,\hat{\circ}\, \alpha \to \beta \cdot x \,\hat{\circ}\, \alpha \rangle\!\rangle$. Both these expressions are well typed according to the definition in figure 6, but only the first could be constructed by the type-inference system of figure 7. Accordingly, *subject reduction* holds of *reFLect* only with respect to the notion of being well-typed, not the stronger property of being type inferable.

Note that while the rules in Figure 7 require variables (in the same scope) in a common quotation to share a common type, they do not require variables with the same name in different quotations to share a type. For example, the type inference system may construct the well typed expression $f \cdot \langle\!\langle 1 + x \rangle\!\rangle \cdot \langle\!\langle \mathsf{T} \wedge x \rangle\!\rangle$.

We could avoid the construction of expressions with multiple variables of the same name and different type if for quoted expressions we retained not only the information about the type of the expression, but also the type-checking environment describing the types of the variables it contains. For antiquotations we would record not only the expected type, but also the prevailing type-checking environment, which describes expectations about the types of incoming variables. The operation to splice one expression into another could then complete a conventional type inference operation on the entire expression.

This approach is not, however, appropriate for our applications in theorem proving. Consider the standard logical rule for conjunction introduction:

$$\frac{\vdash P \quad \vdash Q}{\vdash P \wedge Q}$$

An implementation of this rule is straightforward in *reFLect* using quoted expressions. In the rule, P and Q stand for two separate and arbitrary boolean expressions, perhaps with free variables. Logically, the rule is valid even if P and Q contain variables with the same name but different types.

It would complicate the presentation and use of the logic if rules like this were restricted with side-conditions to ensure the consistent typing of variables in the result. The decision to allow well typed expressions containing variables with the same name and different types is one that *reFLect* shares with the object languages of more conventional theorem proving systems for typed logics, such as HOL.

4.2.4 Extending Static Typing

It is possible to design a more elaborate static typing system than the one just described, with the aim of catching more type errors before runtime. For example, it shouldn't really be necessary to wait until runtime to find out that $\langle\!\langle 1 + \hat{\ }\langle\!\langle \mathsf{T} \rangle\!\rangle \rangle\!\rangle$ is going to run into trouble. The only concern is that this extension might complicate the static semantics of the language. The next paragraph shows some less obvious examples that an extended static type system might detect.

Consider the expression $(\langle\!\langle 1 + \hat{\ }x \rangle\!\rangle, \langle\!\langle \mathsf{T} \wedge \hat{\ }x \rangle\!\rangle)$. This expression is statically well-typed, but at runtime we can already see that it will fail, because no runtime value of x could contain simultaneously both an integer and a boolean. Now consider the expression $(\langle\!\langle 1 :: \hat{\ }x \rangle\!\rangle, \langle\!\langle \mathsf{T} :: \hat{\ }x \rangle\!\rangle)$. This expression looks like it might be dynamically type correct as x may be bound to $\langle\!\langle [] \,\hat{\circ}\, \alpha\ \mathit{list} \rangle\!\rangle$. However, the reduction rules presented later in Section 6 will not allow the α type variables in the two copies of this expression to be instantiated, and so this too will result in a runtime type failure.

$$
\begin{array}{lcl}
{\_}^{\circ}_{\circ}\,\sigma_{n\phi} & = & \begin{cases} {\_}^{\circ}_{\circ}\,\sigma & \text{, if } n > 0 \\ {\_}^{\circ}_{\circ}\,\sigma_{\phi} & \text{, if } n = 0 \end{cases} \\
k^{\circ}_{\circ}\,\sigma_{n\phi} & = & \begin{cases} k^{\circ}_{\circ}\,\sigma & \text{, if } n > 0 \\ k^{\circ}_{\circ}\,\sigma_{\phi} & \text{, if } n = 0 \end{cases} \\
v^{\circ}_{\circ}\,\sigma_{n\phi} & = & \begin{cases} v^{\circ}_{\circ}\,\sigma & \text{, if } n > 0 \\ v^{\circ}_{\circ}\,\sigma_{\phi} & \text{, if } n = 0 \end{cases} \\
(\lambda\mathcal{C}.\,\mathcal{D})_{n\,\phi} & = & \lambda\mathcal{C}_{n\phi}.\,\mathcal{D}_{n\phi} \\
(\lambda\mathcal{C}.\,\mathcal{D} \mid \mathcal{E})_{n\,\phi} & = & \lambda\mathcal{C}_{n\phi}.\,\mathcal{D}_{n\phi} \mid \mathcal{E}_{n\phi} \\
(\mathcal{C}{\cdot}\mathcal{D})_{n\,\phi} & = & \mathcal{C}_{n\phi}{\cdot}\mathcal{D}_{n\phi} \\
\langle\!\langle\mathcal{C}\rangle\!\rangle_{n\,\phi} & = & \langle\!\langle\mathcal{C}_{n+1\,\phi}\rangle\!\rangle \\
(\hat{\ }\mathcal{C}^{\circ}_{\circ}\,\sigma)_{n\,\phi} & = & \hat{\ }\mathcal{C}_{n-1\,\phi}{}^{\circ}_{\circ}\,\sigma \quad \text{, if } n > 0
\end{array}
$$

Figure 8: Type Instantiation of a Context, $n \geq 0$

4.2.5 Type Instantiation of Contexts

We may apply a type instantiation to a context by instantiating every type that appears at level zero in the context. We write $\mathcal{C}_{n\phi}$ to indicate the result of applying the type instantiation ϕ to the context (or expression) $\mathcal{C}$ at level n. In the case where n is zero we will simply write $\mathcal{C}_{\phi}$. Type instantiation of a context is defined in Figure 8.

PROPOSITION 7. *If* $0 \vdash \Lambda$ *and* $\vdash \Lambda{:}\,\sigma$*, then for any type instantiation* ϕ $0 \vdash \Lambda_{\phi}$ *and* $\vdash \Lambda_{\phi}{:}\,\sigma_{\phi}$

5. ABSTRACTIONS

Abstractions in *reFLect* are more complex than in the λ-calculus because any expression may appear in the binding position. This complicates our notion of variable binding and therefore our notion of substitution. Binding and substitution are further complicated by the notion of level. This section describes binding and substitution, and gives an informal introduction to the meaning of abstraction in *reFLect*.

5.1 Binding

An abstraction in the λ-calculus is an expression of the form $\lambda v.\,\Lambda$. The free variables of this expression are the free variables of Λ except for v, which the expression is said to *bind*. Let us ignore the presence of quotation and antiquotation in *reFLect* for a moment and imagine a language that allowed abstractions of the form $\lambda\Lambda.\,\mathrm{M}$. We will say that the free variables of this expression are the free variables of M except for the free variables of Λ.

We now consider the effect of level on binding. Consider $\lambda v.\,v + 1$ and $\lambda w.\,1 + w$. These expressions have different syntax, but they denote the same semantic object, namely the function that increments its argument. Now consider $\langle\!\langle\lambda v.\,v + 1\rangle\!\rangle$ and $\langle\!\langle\lambda w.\,1 + w\rangle\!\rangle$. These expressions denote different semantic objects, namely the syntax of two programs that compute the increment function in different ways. In fact, we even consider the expressions $\langle\!\langle\lambda v.\,v\rangle\!\rangle$ and $\langle\!\langle\lambda w.\,w\rangle\!\rangle$ to be different. They denote semantic objects that represent the syntax of different programs, albeit different programs that both compute the identity function.

In *reFLect*, therefore, the expressions $\lambda v.\,v$ and $\lambda w.\,w$ are equal while $\langle\!\langle\lambda v.\,v\rangle\!\rangle$ and $\langle\!\langle\lambda w.\,w\rangle\!\rangle$ are not. The unquoted λs in the first pair of expressions act as binders, but the quoted λs in the second pair of expressions do not; they act like syntax constructors. This allows us to write functions that construct lambda expressions. Consider β-reducing the expression $(\lambda v.\,\langle\!\langle\lambda\hat{\ }v.\,\hat{\ }v + 1\rangle\!\rangle){\cdot}\langle\!\langle w\rangle\!\rangle$ to $\langle\!\langle\lambda\hat{\ }\langle\!\langle w\rangle\!\rangle.\,\hat{\ }\langle\!\langle w\rangle\!\rangle + 1\rangle\!\rangle$. Section 6.1 will explain how this expression may be reduced to $\langle\!\langle\lambda w.\,w + 1\rangle\!\rangle$. We can think of the *reFLect* expression $\langle\!\langle\lambda\hat{\ }t.\,\hat{\ }u\rangle\!\rangle$ as a meta-language program that constructs an object-level abstraction. Viewed from this perspective, t is free in this expression, at least at the meta-level, but any variables in the value t takes on will be bound at the object level in the result.

$$
\begin{array}{lcl}
\mathit{free}\ k^{\circ}_{\circ}\,\sigma & = & \{\} \\
\mathit{free}\ v^{\circ}_{\circ}\,\sigma & = & \{v^{\circ}_{\circ}\,\sigma\} \\
\mathit{free}\ \lambda\Lambda.\,\mathrm{M} & = & \mathit{free}\ \mathrm{M} - \mathit{free}\ \Lambda \\
\mathit{free}\ \lambda\Lambda.\,\mathrm{M} \mid \mathrm{N} & = & \mathit{free}\ \lambda\Lambda.\,\mathrm{M} \cup \mathit{free}\ \mathrm{N} \\
\mathit{free}\ \Lambda{\cdot}\mathrm{M} & = & \mathit{free}\ \Lambda \cup \mathit{free}\ \mathrm{M} \\
\mathit{free}\langle\!\langle\mathcal{C}[\hat{\ }\Lambda_1{}^{\circ}_{\circ}\,\sigma_1, \ldots \hat{\ }\Lambda_n{}^{\circ}_{\circ}\,\sigma_n]\rangle\!\rangle & = & \mathit{free}\ \Lambda_1 \cup \ldots \mathit{free}\ \Lambda_n \\
\multicolumn{3}{c}{(\text{where } 0 \vdash \mathcal{C}[{\_}^{\circ}_{\circ}\,\sigma_1, \ldots {\_}^{\circ}_{\circ}\,\sigma_n])}
\end{array}
$$

Figure 9: Free Variables

The approach we take is to consider only those variables that appear at level zero in the binding position of a level zero abstraction to be bound. For example, consider the expression $\lambda\langle\!\langle\hat{\ }x + \hat{\ }y\rangle\!\rangle.\,\langle\!\langle\hat{\ }y + \hat{\ }x\rangle\!\rangle$, which binds x and y. This denotes a function that pattern matches quoted additions and commutes them. In contrast, consider the expression $\lambda\langle\!\langle x + y\rangle\!\rangle.\,\langle\!\langle y + x\rangle\!\rangle$, which binds no variables. This denotes a function that pattern matches quoted additions where the first argument literally is 'x' and the second argument literally is 'y', and always returns the quoted addition $\langle\!\langle y + x\rangle\!\rangle$. Patterns with fixed variable names—like this last one—don't appear useful, but they have application in searching for specific variables in a large expression. Figure 9 shows the definition of a function *free*, which describes the free variables of an expression.

5.1.1 Binding and Level

An alternative binding scheme would allow abstractions to bind variables at equal or higher level. In such a system the expression $(\lambda x.\,\langle\!\langle x\rangle\!\rangle){\cdot}1$ would evaluate to $\langle\!\langle 1\rangle\!\rangle$. This binding scheme is used in MetaML, where it is called *cross-stage persistence*.

Cross-stage persistence is not appropriate for the object language of a theorem prover for standard logics. Consider the formula $\neg(\langle\!\langle x\rangle\!\rangle = \langle\!\langle 1\rangle\!\rangle)$. This statement seems transparently true, and indeed *reFLect* evaluates this expression to true. We desire this behavior because we want to write programs that distinguish between the syntax of an object-language variable x and the syntax of an object-language constant 1. But if quantifiers were to bind variables at higher levels then we could make the following sequence of deductions using standard logical quantifier rules, leading to an inconsistent logic.

$$
\dfrac{\dfrac{\vdash \neg(\langle\!\langle x\rangle\!\rangle = \langle\!\langle 1\rangle\!\rangle)}{\vdash \forall x.\,\neg(\langle\!\langle x\rangle\!\rangle = \langle\!\langle 1\rangle\!\rangle)}}{\vdash \neg(\langle\!\langle 1\rangle\!\rangle = \langle\!\langle 1\rangle\!\rangle)}
$$

Suppes [31] also observes this problem and concludes that 'Rule (II) [the prohibition on binding at higher levels] ... is to be abandoned only for profound reasons.' Taha [32] observes the same problem from the perspective of including intensional analysis in MetaML. He notes, as we do, that intensional analysis requires reductions to be allowed only

at level zero, but that this restriction cannot be enforced in a language with cross-stage persistence without loss of confluence.

5.2 The Meaning of Abstractions

The expression that occurs in the binding position of an abstraction in *reFLect* is treated as a pattern. As discussed above, a pattern may bind several variables simultaneously. A pattern may also be partial, in the sense that it does not match all possible values of the relevant type. For example, the pattern in $\lambda\langle\!\langle \hat{} f \cdot \hat{} x \rangle\!\rangle.\ f$ ranges over only that subset of the type of expressions containing syntactic applications. When applied to an expression outside this subset, the result of this function is unspecified.

Moreover, it is syntactically possible for a pattern to contain several instances of a variable, as in $\lambda\langle\!\langle \hat{} x + \hat{} x \rangle\!\rangle.\ \langle\!\langle 2 * \hat{} x \rangle\!\rangle$. We do not require an implementation to evaluate such expressions; any attempt to do so may cause a run-time error. But because such expressions may occur in a logic based on *reFLect*, we need to take at least an informal position on their semantics, so that basic operations like substitution and type instantiation respect this semantics.

One possible approach to the semantics of duplicate pattern variables is to consider only the rightmost occurrence of a variable in a pattern to bind the variable in the body. Then we would expect $(\lambda\langle\!\langle \hat{} x + \hat{} x \rangle\!\rangle.\ \langle\!\langle 2 * \hat{} x \rangle\!\rangle) \cdot \langle\!\langle 1 + 2 \rangle\!\rangle$ to be semantically equal to $\langle\!\langle 2 * 2 \rangle\!\rangle$. This works for patterns that are essentially terms in a free algebra. However, in *reFLect* any expression can occur in pattern position, so we instead take the position that in the pattern of a function such as $\lambda\langle\!\langle \hat{} x + \hat{} x \rangle\!\rangle.\ \langle\!\langle 2 * \hat{} x \rangle\!\rangle$ *both* occurrences of x bind the variable x in the body. The pattern then places a constraint on which applications of the function can be reduced. In this example, the constraint is that the expression to which the function must be an additions of two syntactically identical expressions. Hence we expect $(\lambda\langle\!\langle \hat{} x + \hat{} x \rangle\!\rangle.\ \langle\!\langle 2 * \hat{} x \rangle\!\rangle) \cdot \langle\!\langle 1 + 1 \rangle\!\rangle$ to be semantically equal to $\langle\!\langle 2 * 1 \rangle\!\rangle$. If the constraint is not satisfied, then application of the function is not defined.

In the HOL logic, we would usually express this kind of partially-defined object as an 'under-specified' total function [24]. Formally, one uses a *selection* operator [21] to construct an expression '$\varepsilon\, x.\, P[x]$' with the meaning 'an x such that $P[x]$, or a fixed but unknown value if no such x exists'. With this approach, we can view the abstraction $\lambda\Lambda.\,\mathrm{M}$ as an abbreviation for

$$\varepsilon f.\, \forall\, \mathsf{free}\, \Lambda.\ f\Lambda = \mathrm{M}$$

For example, $\lambda(x, y).\, y$ is the function $\varepsilon f.\, \forall x\, y.\, f(x, y) = y$. We may then view $\lambda\Lambda.\,\mathrm{M} \mid \mathrm{N}$ as an abbreviation for

$$\lambda v.\ \mathsf{if}\ (\forall\, \mathsf{free}\, \Lambda.\ v \neq \Lambda)\ \mathsf{then}\ \mathrm{N}\ v\ \mathsf{else}\ (\lambda\Lambda.\,\mathrm{M})\ v$$

where the variable v is chosen to be distinct from all variables in $\mathsf{free}\{\Lambda, \mathrm{M}, \mathrm{N}\}$.

5.3 Substitution and Type Instantiation

Substitution and type instantiation in *reFLect* are a little more complex than in the λ-calculus, owing to the presence of pattern matching. The two operations are defined as follows.

5.3.1 Substituting Expressions

$$\begin{array}{lcl}
k \circ \sigma\theta & = & k \circ \sigma \\
v \circ \sigma\theta & = & \theta(v \circ \sigma) \\
(\lambda\Lambda.\,\mathrm{M})\theta & = & \lambda\Lambda\iota.\,\mathrm{M}\iota\theta \\
(\lambda\Lambda.\,\mathrm{M} \mid \mathrm{N})\theta & = & \lambda\Lambda\iota.\,\mathrm{M}\iota\theta \mid \mathrm{N}\theta \\
(\Lambda \cdot \mathrm{M})\theta & = & \Lambda\theta \cdot \mathrm{M}\theta \\
\langle\!\langle \mathcal{C}[\hat{}\Lambda_1 \circ \sigma_1, \ldots \hat{}\Lambda_n \circ \sigma_n] \rangle\!\rangle\theta & = & \langle\!\langle \mathcal{C}[\hat{}\Lambda_1\theta \circ \sigma_1, \ldots \hat{}\Lambda_n\theta \circ \sigma_n] \rangle\!\rangle \\
\multicolumn{3}{l}{\quad (\text{where } 0 \vdash \mathcal{C}[_ \circ \sigma_1, \ldots _ \circ \sigma_n] \text{ and}} \\
\multicolumn{3}{l}{\quad\ \iota \text{ is a renaming such that:}} \\
\multicolumn{3}{l}{\quad\quad \mathsf{dom}\,\iota \subseteq \mathsf{free}\,\Lambda, \text{ and } \mathsf{dom}\,\theta \cap \iota(\mathsf{free}\,\Lambda) = \{\}, \text{ and}} \\
\multicolumn{3}{l}{\quad\quad (\mathsf{free}\,\mathrm{M} - \mathsf{free}\,\Lambda) \cap \iota(\mathsf{dom}\,\iota) = \{\}, \text{ and}} \\
\multicolumn{3}{l}{\quad\quad \mathsf{free}(\theta(\mathsf{free}\,\mathrm{M} - \mathsf{free}\,\Lambda)) \cap \iota(\mathsf{free}\,\Lambda) = \{\})}
\end{array}$$

Figure 10: Substitution

A *substitution* is a mapping from variables to expressions of the same type that is the identity on all but finitely many variables. We typically use the meta-variables θ and ι to stand for substitutions. We write $\mathsf{dom}\,\theta$ for the *domain* of θ, meaning the set of variables for which θ is not the identity. If $\mathsf{dom}\,\theta = \{v_1 \circ \sigma_1, \ldots v_n \circ \sigma_n\}$ and $\theta(v_i \circ \sigma_i) = \Lambda_i$ for all $1 \leq i \leq n$, then we sometimes write θ using the notation $[\Lambda_1, \ldots \Lambda_n / v_1 \circ \sigma_1, \ldots v_n \circ \sigma_n]$. A *renaming* is an injective substitution that maps variables to variables.

For any expression Λ and substitution θ we may write $\Lambda\theta$ to stand for the action of applying the substitution to all the free variables of Λ, with appropriate renaming of the bound variables in Λ to avoid capture. Figure 10 defines this operation.[2]

Note that substitution must be consistent with the interpretation we place on repeated pattern variables. We require the result of $(\lambda(x, x).\, y)\,[x/y]$ to be $\lambda(x', x').\, x$. That is, both occurrences of x in the pattern are renamed.

PROPOSITION 8. *If* $0 \vdash \Lambda$ *and* $\vdash \Lambda : \sigma$*, then for any substitution* θ $0 \vdash \Lambda\theta$ *and* $\vdash \Lambda\theta : \sigma$*.*

Most HOL-style theorem provers have a more general substitution primitive, which allows one to substitute for arbitrary *subexpressions* occurring free in an expression, not just for free variables. This is also the case in the *reFLect* theorem prover, but variable-substitution suffices for presenting the operational semantics.

5.3.2 Type Instantiation

We may also apply a type instantiation to an expression. For any expression Λ and type instantiation ϕ, we write $\Lambda\phi$ to mean the result of applying the instantiation to the expression. This applies the instantiation to every level zero type in the expression, using the notion of instantiation defined in Section 3.1.1. Since the identity of a variable in *reFLect* consists of its name and type, we need to rename bound variables to avoid capture during a type instantiation. For example, $(\lambda(x \circ \alpha, x \circ \beta).\, x \circ \alpha)[\beta/\alpha]$ should produce $\lambda(x' \circ \beta, x \circ \beta).\, x' \circ \beta$ or $\lambda(x \circ \beta, x' \circ \beta).\, x \circ \beta$.

The formal definition of type instantiation for expressions is similar to the definition of substitution in Figure 10. Note that $\Lambda\phi$ is not the same as the context type instantiation

[2] In the condition of figure 10, ι, θ, and free are implicitly extended to image functions over sets where required.

$$\frac{\vdash \mathit{definition}\, k\colon \tau \quad \tau_\phi = \sigma}{\vdash k \circ \sigma \to (\mathit{definition}\, k)\phi}\,[\delta]$$

$$\frac{\mathit{pattern}\,\Lambda \quad \Lambda\ \mathit{ready}\ \Xi \quad (\Lambda,\theta)\ \mathit{matches}\ \Xi}{\vdash (\lambda\Lambda.\,\mathrm{M})\cdot\Xi \to \mathrm{M}\theta}\,[\beta]$$

$$\frac{\mathit{pattern}\,\Lambda \quad \Lambda\ \mathit{ready}\ \Xi \quad (\Lambda,\theta)\ \mathit{matches}\ \Xi}{\vdash (\lambda\Lambda.\,\mathrm{M} \mid \mathrm{N})\cdot\Xi \to \mathrm{M}\theta}\,[\gamma]$$

$$\frac{\mathit{pattern}\,\Lambda \quad \Lambda\ \mathit{ready}\ \Xi \quad \nexists\theta.\,(\Lambda,\theta)\ \mathit{matches}\ \Xi}{\vdash (\lambda\Lambda.\,\mathrm{M} \mid \mathrm{N})\cdot\Xi \to \mathrm{N}\cdot\Xi}\,[\zeta]$$

$$\frac{0 \vdash \mathcal{C}[\_\circ\sigma_1, \ldots \_\circ\sigma_n] \quad \vdash \Lambda_1\colon\sigma_{1\phi} \quad \ldots \quad \vdash \Lambda_n\colon\sigma_{n\phi} \quad \mathit{dom}\,\phi \subseteq \mathit{vars}\{\sigma_1, \ldots \sigma_n\}}{\vdash \langle\!\langle \mathcal{C}[\hat{}\langle\!\langle\Lambda_1\rangle\!\rangle\circ\sigma_1, \ldots \hat{}\langle\!\langle\Lambda_n\rangle\!\rangle\circ\sigma_n]\rangle\!\rangle \to \langle\!\langle \mathcal{C}_\phi[\Lambda_1, \ldots \Lambda_n]\rangle\!\rangle}\,[\psi]$$

Figure 11: Reduction

operation $\mathcal{C}_\phi$ in Figure 8, which does not rename variables to avoid capture. We will not use type instantiation on expressions as described here until section 7.1.

6. OPERATIONAL SEMANTICS

Figures 11 and 12 present the reduction rules for evaluating a *reFLect* expression. The rules in Figure 11 describe individual reductions, while those in Figure 12 describe how reductions may be applied to subexpressions. The judgments are of the form $\vdash \Lambda \to \Lambda'$, which means that Λ reduces to Λ' in one step. These rules ensure that reductions apply only to level zero subexpressions, and then only to those that do not fall in the binding position of a level zero abstraction. We use the standard notation $\vdash \Lambda \xrightarrow{*} \Lambda'$ to indicate that Λ can be reduced to Λ' in zero or more steps. This is formalized in figure 13.

The rules of Figure 11 use some auxiliary meta-functions, which we briefly introduce here and describe in more detail later. The function *definition* returns the definition of a constant. The predicate *pattern* characterizes the expressions we consider valid for pattern matching against, variables or quotations whose level zero subexpressions are variables. The relation (Λ,θ) *matches* Ξ means that applying the substitution θ to the pattern Λ causes it to match the expression Ξ (in a sense we define precisely later). The relation Λ *ready* M means that the expression M has been sufficiently evaluated to determine whether or not it matches the pattern Λ.

PROPOSITION 9. *If* $0 \vdash \Lambda$ *and* $\vdash \Lambda\colon\sigma$, *then for any* M *such that* $\vdash \Lambda \xrightarrow{*} \mathrm{M}$ *we have* $0 \vdash \mathrm{M}$ *and* $\vdash \mathrm{M}\colon\sigma$.

Proposition 9 is the subject reduction property for *reFLect*. The property states that a level consistent and well typed expression remains so as it is reduced, and that the expression retains the same type as it is reduced. Krstić and Matthews have a proof of this property [20].

6.1 Reducing Quotations

The rule for ψ-reduction in Figure 11 allows the elimination of antiquoted quotations at level one. The rule caters for the possibility that the type variables of a quoted region may need to be instantiated in order to be type consistent

$$\frac{\vdash \mathrm{M} \to \mathrm{M}'}{\vdash \lambda\Lambda.\,\mathrm{M} \to \lambda\Lambda.\,\mathrm{M}'}$$

$$\frac{\vdash \mathrm{M} \to \mathrm{M}'}{\vdash \lambda\Lambda.\,\mathrm{M} \mid \mathrm{N} \to \lambda\Lambda.\,\mathrm{M}' \mid \mathrm{N}} \qquad \frac{\vdash \mathrm{N} \to \mathrm{N}'}{\vdash \lambda\Lambda.\,\mathrm{M} \mid \mathrm{N} \to \lambda\Lambda.\,\mathrm{M} \mid \mathrm{N}'}$$

$$\frac{\vdash \Lambda \to \Lambda'}{\vdash \Lambda\cdot\mathrm{M} \to \Lambda'\cdot\mathrm{M}} \qquad \frac{\vdash \mathrm{M} \to \mathrm{M}'}{\vdash \Lambda\cdot\mathrm{M} \to \Lambda\cdot\mathrm{M}'}$$

$$\frac{0 \vdash \mathcal{C}[\_\circ\sigma_1, \ldots \_\circ\sigma_m, \ldots \_\circ\sigma_n] \quad \vdash \Lambda_m \to \Lambda'_m}{\vdash \langle\!\langle \mathcal{C}[\hat{}\Lambda_1\circ\sigma_1, \ldots \hat{}\Lambda_m\circ\sigma_m, \ldots \hat{}\Lambda_n\circ\sigma_n]\rangle\!\rangle \to \langle\!\langle \mathcal{C}[\hat{}\Lambda_1\circ\sigma_1, \ldots \hat{}\Lambda'_m\circ\sigma_m, \ldots \hat{}\Lambda_n\circ\sigma_n]\rangle\!\rangle}$$

Figure 12: Reducing Subexpressions

$$\frac{}{\vdash \Lambda \xrightarrow{*} \Lambda} \qquad \frac{\vdash \Lambda \to \mathrm{M} \quad \vdash \mathrm{M} \xrightarrow{*} \mathrm{N}}{\vdash \Lambda \xrightarrow{*} \mathrm{N}}$$

Figure 13: Reduction Closure

with the antiquoted regions being spliced into it. Suppose, for example, that *inc* is a constant of type $\mathit{int} \to \mathit{int}$. The ψ rule lets us reduce $\langle\!\langle \hat{}\langle\!\langle \mathit{inc}\rangle\!\rangle \circ \alpha \to \beta\cdot\hat{}\langle\!\langle 1\rangle\!\rangle \circ \alpha\rangle\!\rangle$ to $\langle\!\langle \mathit{inc}\cdot 1\rangle\!\rangle$ by allowing α and β to be instantiated to *int*.

This type-instantiation behavior of ψ is the basis for run-time type checking in *reFLect*. At compile time, we type-check quotation contexts at their most general types. Then at run-time—when the expressions being spliced into the holes become available—we check type consistency by instantiating the context's type variables to match the types inside the incoming expressions. For example, consider the function *comm* in Section 2. Static type checking will assign polymorphic types to the quotations in the definition of *comm* so that, for example, $\mathit{comm}\cdot\langle\!\langle \mathit{inc}\cdot(1+2)\rangle\!\rangle$ reduces at run time to $\langle\!\langle \hat{}\langle\!\langle \mathit{inc}\rangle\!\rangle \circ \alpha \to \beta\cdot\hat{}\langle\!\langle 2+1\rangle\!\rangle \circ \alpha\rangle\!\rangle$. Then, using ψ-reduction, we get the expected expression $\langle\!\langle \mathit{inc}\cdot(2+1)\rangle\!\rangle$.

The rule does not allow reductions to create badly-typed expressions. For example, we cannot use this rule to reduce the expression $\langle\!\langle \hat{}\langle\!\langle \mathit{inc}\rangle\!\rangle \circ \alpha \to \beta\cdot\hat{}\langle\!\langle \mathsf{T}\rangle\!\rangle \circ \alpha\rangle\!\rangle$. Note also that the rule does not allow type instantiations of the expressions inside the antiquotes. For example, we cannot use this rule to reduce the expression $\langle\!\langle \hat{}\langle\!\langle f \circ \alpha \to \beta\rangle\!\rangle \circ \mathit{int} \to \mathit{int}\cdot 1\rangle\!\rangle$.

6.1.1 Instantiation Must Affect the Entire Term

One might first imagine a simpler rule for ψ-reduction like the one shown below:

$$\frac{\vdash \Lambda\colon\tau_\phi}{\vdash \hat{}\langle\!\langle\Lambda\rangle\!\rangle \circ \tau \to \Lambda}$$

Unfortunately the effect of this rule does not cover enough of the expression to ensure type consistency. Consider again the expression $\langle\!\langle \hat{}\langle\!\langle \mathit{inc}\rangle\!\rangle \circ \alpha \to \beta\cdot\hat{}\langle\!\langle \mathsf{T}\rangle\!\rangle \circ \alpha\rangle\!\rangle$. We could use this incorrect rule to reduce it to $\langle\!\langle \mathit{inc} \circ \mathit{int} \to \mathit{int}\cdot\hat{}\langle\!\langle \mathsf{T}\rangle\!\rangle \circ \alpha\rangle\!\rangle$ and then again to $\langle\!\langle \mathit{inc} \circ \mathit{int} \to \mathit{int}\cdot\mathsf{T} \circ \mathit{bool}\rangle\!\rangle$.

6.1.2 All Antiquotes Eliminated Simultaneously

The assumption $0 \vdash \mathcal{C}[\_\circ\sigma_1, \ldots \_\circ\sigma_n]$ of the ψ-reduction rule ensures that it eliminates every level one antiquote enclosed by a given quotation. We could imagine a version of

this rule that need not eliminate every antiquote simultaneously. We could then reduce $\langle\!\langle \hat{}\langle\!\langle inc \rangle\!\rangle \colon \alpha \to \beta \cdot \hat{}\langle\!\langle 1 \rangle\!\rangle \colon \alpha \rangle\!\rangle$ to $\langle\!\langle inc \cdot \hat{}\langle\!\langle 1 \rangle\!\rangle \colon int \rangle\!\rangle$ and later to $\langle\!\langle inc \cdot 1 \rangle\!\rangle$. But this rule would also allow us to reduce $\langle\!\langle \hat{}\langle\!\langle inc \rangle\!\rangle \colon \alpha \to \beta \cdot \hat{}\langle\!\langle \mathsf{T} \rangle\!\rangle \colon \alpha \rangle\!\rangle$ to both $\langle\!\langle inc \cdot \hat{}\langle\!\langle \mathsf{T} \rangle\!\rangle \colon int \rangle\!\rangle$ and $\langle\!\langle \hat{}\langle\!\langle inc \rangle\!\rangle \colon int \to \beta \cdot \mathsf{T} \rangle\!\rangle$. Since these expressions may not be further reduced this would leave *reFLect* with a non-confluent reduction system.[3]

We could, however, allow a rule that requires only that all the antiquotes occurring *at the same level* within a given quoted region need be eliminated simultaneously. For example, consider

$$\langle\!\langle (\hat{}\langle\!\langle 1 \rangle\!\rangle, \hat{}\langle\!\langle 2 \rangle\!\rangle, \langle\!\langle (\tilde{}\langle\!\langle \langle\!\langle 3 \rangle\!\rangle \rangle\!\rangle, \tilde{}\langle\!\langle \langle\!\langle 4 \rangle\!\rangle \rangle\!\rangle) \rangle\!\rangle) \rangle\!\rangle$$

The first two antiquotes must be eliminated simultaneously, and so must the second two, but it would be possible to develop a valid semantics that did not require all four to be eliminated together. Expressions like this, however, do not arise in our applications—so we do not complicate the semantics to facilitate this relaxation.

6.1.3 Type Instantiation Impacts Only the Context

The ψ-reduction operation ensures that it constructs a well typed expression by type instantiating the context into which the antiquoted expressions are spliced. One might also consider *unifying* the types of the context and the incoming expressions to achieve a match. This is the approach taken in the system of Shields et al. [29].

This option was rejected for reasons that derive from the target application of *reFLect* to theorem proving and circuit transformation. In these applications most operations that manipulate expressions are expected to preserve the types of the manipulated expressions. In this case, unification is not appropriate. This is in contrast to systems designed for code-generation [28] or staged evaluation [33], which focus more on flexible ways of constructing or specializing programs.

For example, a ubiquitous theorem proving application is term rewriting [25], in which an expression is transformed by application of general rewrite rules to its subexpressions. The matching that makes a general rewrite rule applicable at a subexpression is always one-way and type unification is not appropriate. The semantics of our hole-filling ψ rule therefore exactly achieves the *reFLect* design requirement for a native mechanism to support rewriting.

In theorem proving and transformation applications, contexts are typically small and the incoming expressions very large. The same expression may also be spliced into more than one context. If we unified types when splicing an expression into a context, we could not do it by destructively instantiating type variables, a constant time operation. Rather, we would have to copy incoming expressions using time and space proportionate to their size. Since the speed of rewriting is key to the effectiveness of a theorem prover, we would not be able to use this splicing operation to implement our rewriter.

6.2 Patterns May Not Be Reduced

An examination of the rules in Figure 12 reveals that it is possible to reduce any level zero subexpression, except those in the binding position of an abstraction. Patterns may not be reduced. We might imagine a system that allowed reductions on patterns as well. For example, it seems reasonable to reduce the expression $(\lambda\langle\!\langle \hat{}\langle\!\langle 1 \rangle\!\rangle + \hat{}x \rangle\!\rangle . x) \cdot \langle\!\langle 1 + 2 \rangle\!\rangle$ to $(\lambda\langle\!\langle 1 + \hat{}x \rangle\!\rangle . x) \cdot \langle\!\langle 1 + 2 \rangle\!\rangle$ and then to $\langle\!\langle 2 \rangle\!\rangle$.

But unrestricted reduction of patterns is unsafe. As an example, consider the expression $\lambda(\lambda y.\, z) \cdot x.\, x$, in which the pattern $(\lambda y.\, z) \cdot x$ occurs in binding position. If we were to allow reduction of this pattern, we could reduce the whole expression to $\lambda z.\, x$. But then the variable x, which was bound in the original expression, has become free—perhaps to be captured by some enclosing scope. It might be possible to avoid this problem by not allowing pattern reductions that change the free variable set of the pattern. But in the absence of a compelling application, it seems simpler just to forbid all pattern reductions.

$$\frac{}{\mathit{pattern}\ v \colon \sigma}$$

$$\frac{0 \vdash \mathcal{C}[\_ \colon \sigma_1, \ldots \_ \colon \sigma_n]}{\mathit{pattern} \langle\!\langle \mathcal{C}[\hat{}(v_1 \colon term) \colon \sigma_1, \ldots \hat{}(v_n \colon term) \colon \sigma_n] \rangle\!\rangle}$$

Figure 14: Valid Pattern

6.3 Pattern Matching

The rules for β-reduction, γ-reduction, and ζ-reduction apply only to abstractions over *valid* patterns. Not all expressions make valid patterns. For example, the expressions in the binding positions of $\lambda x.\, x$ and $\lambda\langle\!\langle \hat{}x + 1 \rangle\!\rangle . \langle\!\langle 1 + \hat{}x \rangle\!\rangle$ are both valid patterns, but the binding expression in $\lambda x + 1.\, x$ is not. This is not to say that such bindings are without meaning, only that we do not support the evaluation of such patterns, and so they are considered invalid for the purposes of this operational semantics.

Figure 14 defines the predicate *pattern* that characterizes which patterns are considered valid. It can be summarized by saying that a valid pattern is either a variable or a quotation where every level zero subexpression is a variable. The definition does not rule out patterns containing more than one instance of the same variable. An implementation, however, may have a stricter notion of valid pattern that disallows this. Any attempt to match a invalid pattern should lead to a run-time failure.

We also make some restrictions on when we are prepared to consider matching a pattern. If a pattern is a simple variable, then we may match it straightaway, but if a pattern is a quotation then we must wait until the expression we are trying to match has been reduced to a quotation with level one antiquotes eliminated. We will say that the expression M is *ready* to be matched to the pattern Λ, Λ *ready* M, if this condition holds. Figure 15 formalizes this notion, which is used in the rules for β and γ-reduction in figure 11.

Consider what can happen without this restriction by con-

$$\frac{}{v\ \mathit{ready}\ \Lambda} \qquad \frac{0 \vdash \Lambda}{\mathrm{M}\ \mathit{ready}\ \langle\!\langle \Lambda \rangle\!\rangle}$$

Figure 15: Match Readiness

[3]It may still have some property similar to confluence, in which expressions like these are considered equivalent.

$$\frac{v_\circ^\circ\,\sigma\,\theta = \Xi}{(v_\circ^\circ\,\sigma, \theta)\ \mathit{matches}\ \Xi}$$

$$\frac{\begin{array}{l}0 \vdash \mathcal{C}[\__\circ^\circ\,\sigma_1, \ldots \__\circ^\circ\,\sigma_n] \\ \phi \vdash \mathcal{C}[w_1{}_\circ^\circ\,\sigma_1, \ldots w_n{}_\circ^\circ\,\sigma_n] \rightsquigarrow \mathcal{D}[w_1{}_\circ^\circ\,\sigma_{1\phi}, \ldots w_n{}_\circ^\circ\,\sigma_{n\phi}] \\ v_1{}_\circ^\circ\,\mathit{term}\,\theta = \langle\!\langle \Xi_1 \rangle\!\rangle \quad \ldots \quad v_n{}_\circ^\circ\,\mathit{term}\,\theta = \langle\!\langle \Xi_n \rangle\!\rangle\end{array}}{\begin{array}{l}(\langle\!\langle \mathcal{C}[\hat{}(v_1{}_\circ^\circ\,\mathit{term})\,\sigma_1, \ldots \hat{}(v_n{}_\circ^\circ\,\mathit{term})\,\sigma_n] \rangle\!\rangle, \theta)\ \mathit{matches} \\ \langle\!\langle \mathcal{D}[\Xi_1, \ldots \Xi_n] \rangle\!\rangle \\ \quad \text{(where } w_1, \ldots \ w_n \text{ are fresh)}\end{array}}$$

Figure 16: Pattern Matching an Expression

templating the effect of *dest_apply* from section 2. If we apply this to the expression $\langle\!\langle g_\circ^\circ\,\alpha \to \alpha\cdot\hat{}\langle\!\langle 1 \rangle\!\rangle_\circ^\circ\,\alpha \rangle\!\rangle$ and we were to evaluate the application before ψ-reducing the argument we would get the result $(\langle\!\langle g_\circ^\circ\,\alpha \to \alpha \rangle\!\rangle, \langle\!\langle \hat{}\langle\!\langle 1 \rangle\!\rangle_\circ^\circ\,\alpha \rangle\!\rangle)$, which would then reduce to $(\langle\!\langle g_\circ^\circ\,\alpha \to \alpha \rangle\!\rangle, \langle\!\langle 1 \rangle\!\rangle)$. If we were to ψ-reduce the argument before reducing the application we would get the result $(\langle\!\langle g_\circ^\circ\,\mathit{int} \to \mathit{int} \rangle\!\rangle, \langle\!\langle 1 \rangle\!\rangle)$.

As with the possible generalization to ψ-reduction discussed in Section 6.1.2, we believe there is an equally valid semantics that doesn't force the elimination of all level one antiquotes from an expression before it may be matched, but only those from level contiguous regions that are in some way accessed by the match. But this would complicate the semantics without benefit to practical applications.

6.3.1 Matching An Alternative

Once we have determined that a pattern is valid and an expression is ready to be matched by it then we are ready to determine whether (and how) the expression matches the pattern. The predicate *matches*, defined in Figure 16, makes this determination.

When the pattern is a variable, we say that the pattern matches an expression under a substitution precisely when the substitution maps that variable to the expression. When the pattern is a quotation, we first find a level-consistent context $\mathcal{C}[\__\circ^\circ\,\sigma_1, \ldots \__\circ^\circ\,\sigma_n]$ and term variables $v_1, \ldots\ v_n$ such that the pattern we are trying to match against is

$$\langle\!\langle \mathcal{C}[\hat{}(v_1{}_\circ^\circ\,\mathit{term})\,\sigma_1, \ldots \hat{}(v_n{}_\circ^\circ\,\mathit{term})\,\sigma_n] \rangle\!\rangle$$

Next we must find a level consistent context $\mathcal{D}[\__\circ^\circ\,\tau_1, \ldots \__\circ^\circ\,\tau_n]$ and list of subexpressions $\Xi_1, \ldots\ \Xi_n$ such that the expression we are trying to match is $\langle\!\langle \mathcal{D}[\Xi_1, \ldots \Xi_n] \rangle\!\rangle$. The expression matches the pattern if $\mathcal{D}$ is a type instance, in the sense explained below, of $\mathcal{C}$ and we can match each expression Ξ_1, $\ldots\ \Xi_n$ to the corresponding variable $v_1, \ldots\ v_n$ under the same substitution.

The notation $\phi \vdash \Lambda \rightsquigarrow \mathrm{M}$ indicates that M is a type instance of Λ under some type instantiation ϕ and is defined in Figure 17. The role of the $\rightsquigarrow$ relation is to allow the types within quotations in the pattern to be more general than those of the argument. This allows functions on expressions to be defined by pattern matching, as in the following example:

$$\begin{array}{l}\mathsf{let}\ \mathit{len} = \lambda\langle\!\langle \mathsf{Len}\cdot([]_\circ^\circ\,\alpha\ \mathit{list}) \rangle\!\rangle.\ \langle\!\langle 0 \rangle\!\rangle \\ \qquad\quad \mid\ \lambda\langle\!\langle \mathsf{Len}\cdot(\hat{}h :: \hat{}t) \rangle\!\rangle.\ \langle\!\langle (\mathsf{Len}\cdot\hat{}t) + 1 \rangle\!\rangle\end{array}$$

We would expect to be able to apply the first λ-abstraction in this function to expressions such as $\langle\!\langle \mathsf{Len}\cdot([]_\circ^\circ\,\mathit{int}\ \mathit{list}) \rangle\!\rangle$, and so the pattern $\langle\!\langle \mathsf{Len}\cdot([]_\circ^\circ\,\alpha\ \mathit{list}) \rangle\!\rangle$ must match up to some instantiation of type variables.

$$\frac{\sigma_\phi = \tau}{\phi \vdash k_\circ^\circ\,\sigma \rightsquigarrow k_\circ^\circ\,\tau} \qquad \frac{\sigma_\phi = \tau}{\phi \vdash v_\circ^\circ\,\sigma \rightsquigarrow v_\circ^\circ\,\tau}$$

$$\frac{\phi \vdash \Lambda \rightsquigarrow \Lambda' \quad \phi \vdash \mathrm{M} \rightsquigarrow \mathrm{M}'}{\phi \vdash \lambda\Lambda.\,\mathrm{M} \rightsquigarrow \lambda\Lambda'.\,\mathrm{M}'}$$

$$\frac{\phi \vdash \Lambda \rightsquigarrow \Lambda' \quad \phi \vdash \mathrm{M} \rightsquigarrow \mathrm{M}' \quad \phi \vdash \mathrm{N} \rightsquigarrow \mathrm{N}'}{\phi \vdash \lambda\Lambda.\,\mathrm{M} \mid \mathrm{N} \rightsquigarrow \lambda\Lambda'.\,\mathrm{M}' \mid \mathrm{N}'}$$

$$\frac{\phi \vdash \Lambda \rightsquigarrow \Lambda' \quad \phi \vdash \mathrm{M} \rightsquigarrow \mathrm{M}'}{\phi \vdash \Lambda\cdot\mathrm{M} \rightsquigarrow \Lambda'\cdot\mathrm{M}'}$$

$$\frac{\begin{array}{l}\sigma_{1\chi} = \tau_1 \quad \ldots \quad \sigma_{n\chi} = \tau_n \\ \chi \vdash \mathcal{C}[v_1{}_\circ^\circ\,\sigma_1, \ldots v_n{}_\circ^\circ\,\sigma_n] \rightsquigarrow \mathcal{C}'[v_1{}_\circ^\circ\,\tau_1, \ldots v_n{}_\circ^\circ\,\tau_n] \\ 0 \vdash \mathcal{C}[\_1{}_\circ^\circ\,\sigma_1, \ldots \_n{}_\circ^\circ\,\sigma_n] \quad 0 \vdash \mathcal{C}'[\_1{}_\circ^\circ\,\tau_1, \ldots \_n{}_\circ^\circ\,\tau_n] \\ \phi \vdash \Lambda_1 \rightsquigarrow \Lambda'_1 \quad \ldots \quad \phi \vdash \Lambda_n \rightsquigarrow \Lambda'_n\end{array}}{\begin{array}{l}\phi \vdash \langle\!\langle \mathcal{C}[\hat{}\Lambda_1{}_\circ^\circ\,\sigma_1, \ldots \hat{}\Lambda_n{}_\circ^\circ\,\sigma_n] \rangle\!\rangle \rightsquigarrow \langle\!\langle \mathcal{C}'[\hat{}\Lambda'_1{}_\circ^\circ\,\tau_1, \ldots \hat{}\Lambda'_n{}_\circ^\circ\,\tau_n] \rangle\!\rangle \\ \quad \text{(where } v_1 \ \ldots \ v_n \text{ are fresh)}\end{array}}$$

Figure 17: Type-Match Relation

6.3.2 Discarding an Alternative

The rules for β and γ-reduction require the argument expression to be ready to match the pattern before a match is made. Similarly, the rule for ζ-reduction requires the argument expression to be ready to match the pattern before the match is rejected.

In general, we may have $(\lambda\Lambda.\,\mathrm{M} \mid \mathrm{N})\cdot\Xi$, where Ξ has type *term* but has not yet been evaluated to yield a quotation. It is not possible to tell if and how Ξ might match the pattern Λ until Ξ has been evaluated. The assumption Λ *ready* M on the ζ-reduction rule prevents a match from being discarded too early. If an expression M is ready to match a pattern Λ, but there is no substitution θ such that (Λ, θ) *matches* M, then we may safely conclude that the expression doesn't match the pattern and discard this alternative.

In some circumstances a pattern will never match an expression and yet may also not be discarded. Consider the following application:

$$(\lambda\langle\!\langle \hat{}f\cdot\hat{}x \rangle\!\rangle.\ \Lambda \mid \mathrm{M})\cdot\langle\!\langle \hat{}\langle\!\langle \mathit{inc}_\circ^\circ\,\mathit{int} \to \mathit{int} \rangle\!\rangle_\circ^\circ\,\alpha \to \alpha\ \cdot\hat{}\langle\!\langle \mathsf{T} \rangle\!\rangle_\circ^\circ\,\alpha \rangle\!\rangle$$

In this example the argument is not ready to match the pattern, however it may not be further reduced. The *reFLect* language does not let us conclude anything about the internal structure of expressions that are not sufficiently evaluated to tell if they are well typed. In an implementation, the inability to apply either the γ-reduction or ζ-reduction rules would result in the argument being forced to point where ψ-reduction was attempted and a run-time type error raised.

7. REFLECTION

Thus far we have described the core of the *reFLect* language. This language features facilities for constructing and destructing expressions using quotation, antiquotation and pattern matching. These allow *reFLect* to be used for applications, like theorem prover development, that might usually be approached with a system based on a separate meta-language and object-language. In this section we add some facilities for reflection.

7.1 Evaluation

The *reFLect* language has two built-in functions for evaluation of expressions: eval and value.[4] Suppose we use the notation $\Lambda \Rightarrow \Lambda'$ to mean that Λ is evaluated to produce Λ'. Certainly $\Lambda \Rightarrow \Lambda'$ implies $\Lambda \xrightarrow{*} \Lambda'$, but we consider the order of evaluation and the normal form at which evaluation stops to be implementation specific, and so we leave these unspecified. The eval function is then described as follows:

$$\vdash \mathsf{eval}\colon \mathit{term} \to \mathit{term}$$

$$\frac{0 \vdash \Lambda \quad \vdash \Lambda \Rightarrow \Lambda'}{\vdash \mathsf{eval}\cdot\langle\!\langle \Lambda \rangle\!\rangle \to \langle\!\langle \Lambda' \rangle\!\rangle}$$

Next we consider value. It is a slight misstatement to say that value is a function in *reFLect*—rather there is an infinite family of functions value_σ indexed by type. The behavior of value is similar to that of antiquotation; it removes the quotes from around an expression and interprets the result as a value of the appropriate type.

$$\vdash \mathsf{value}_\sigma\colon \mathit{term} \to \sigma$$

$$\frac{0 \vdash \Lambda \quad \mathit{free}\,\Lambda = \emptyset \quad \vdash \Lambda\colon \tau \quad \tau_\phi = \sigma}{\vdash \mathsf{value}_\sigma\cdot\langle\!\langle \Lambda \rangle\!\rangle \to \Lambda\phi}$$

There are several important differences between value and antiquotation:

- The value function may appear at level zero, while antiquotation may not.
- Like other functions, value has no effect when quoted, while a (once) quoted antiquote may be reduced.
- If the type required of an expression is different from the actual type, then value may instantiate the type of the expression. Antiquotation may instead instantiate the type of its context.
- Antiquotation does not alter the level of the quoted expression, but value moves the body of the quoted expression from level one to level zero.

This last difference has important consequences for the treatment of variable binding. In moving an expression to level zero, value could expose its free variables to capture by enclosing lambda bindings. We restrict value to operate on closed expressions to prevent this. The restriction is similar in motivation to the run-time variable check of run in MetaML [33] or the static check for closed code in the system λ^{BN} [4].

7.2 Value Reification

The *reFLect* language also supports a partial inverse of evaluation through the lift function; its purpose is to make quoted representations of values. For example, $\mathsf{lift}\cdot 1$ is $\langle\!\langle 1 \rangle\!\rangle$ and $\mathsf{lift}\cdot\mathsf{T}$ is $\langle\!\langle \mathsf{T} \rangle\!\rangle$. The function lift is strict, so $\mathsf{lift}\cdot(1+2)$ is equal to $\langle\!\langle 3 \rangle\!\rangle$. Note also that lift may only be applied to closed expressions. Lifting quotations is easy: just wrap another quote around them. For example, $\mathsf{lift}\cdot\langle\!\langle x + y \rangle\!\rangle$ gives $\langle\!\langle\langle\!\langle x + y \rangle\!\rangle\!\rangle\rangle$. Lifting recursive data-structures follows a recursive pattern that can be seen from the following example of how lift works on lists.

$$\begin{aligned}&\mathsf{lift}\cdot[]\colon\!\colon \sigma\ \mathit{list} = \langle\!\langle []\colon\!\colon \sigma\ \mathit{list} \rangle\!\rangle\\ &\mathsf{lift}\cdot(::\colon\!\colon \sigma \to \sigma\ \mathit{list} \to \sigma \mathit{list})\cdot\Lambda\cdot\mathrm{M}) =\\ &\quad \langle\!\langle (::\colon\!\colon \sigma \to \sigma\ \mathit{list} \to \sigma \mathit{list})\cdot\hat{}(\mathsf{lift}\cdot\Lambda)\colon\!\colon \sigma\cdot\hat{}(\mathsf{lift}\cdot\mathrm{M})\colon\!\colon \sigma\ \mathit{list} \rangle\!\rangle\end{aligned}$$

Lifting numbers, booleans and recursive data-structures is easy because they have a canonical form, but the same is not true of other data-types. For example, how do we lift $\lambda x.\, x + 1$? Naively wrapping quotations around the expressions would result in inconsistencies. For example, $\lambda x.\, x + 1$ and $\lambda x.\, 1 + x$ are equal and extensionality therefore requires $\mathsf{lift}\cdot(\lambda x.\, x + 1)$ and $\mathsf{lift}\cdot(\lambda x.\, 1 + x)$ to be equal. But the quotations $\langle\!\langle \lambda x.\, x + 1 \rangle\!\rangle$ and $\langle\!\langle \lambda x.\, 1 + x \rangle\!\rangle$ are not equal. If Λ is an expression of some type σ without a canonical form then we will use the following definition for lift.

$$\mathsf{lift}\cdot\Lambda = \langle\!\langle [\![\Lambda]\!]\colon\!\colon \sigma \rangle\!\rangle$$

You should think of $[\![\Lambda]\!]$ as being a new and unusual constant name. These names have the property that if Λ and M are semantically equal, then $[\![\Lambda]\!]$ and $[\![\mathrm{M}]\!]$ are considered the same name. For example evaluating $\mathsf{lift}\cdot(\lambda x.\, x + 1)$ produces $\langle\!\langle [\![\lambda x.\, x + 1]\!] \rangle\!\rangle$ and evaluating $\mathsf{lift}\cdot(\lambda x.\, 1 + x)$ produces $\langle\!\langle [\![\lambda x.\, 1 + x]\!] \rangle\!\rangle$, and the two resulting expressions are equal since they are both quoted constants with 'equal' names.[5] When we do this, we say that we have put the expression in a *black box*. Since black boxes are just a kind of constant they require no further special treatment.

Note that eval is not simply the composition of value and lift. Consider the expressions

$$\langle\!\langle (\lambda x.\, \lambda y.\, x + y)\cdot 1) \rangle\!\rangle \quad \text{and} \quad \langle\!\langle (\lambda x.\, \lambda y.\, y + x)\cdot 1) \rangle\!\rangle$$

Applying eval to these expressions produces $\langle\!\langle \lambda y.\, 1 + y \rangle\!\rangle$ and $\langle\!\langle \lambda y.\, y + 1 \rangle\!\rangle$. Applying value then lift however must yield $\langle\!\langle [\![\lambda y.\, 1 + y]\!] \rangle\!\rangle$ and $\langle\!\langle [\![\lambda y.\, y + 1]\!] \rangle\!\rangle$. Because the composition of value and lift takes an expression to an expression via the unquoted form that represents its meaning, two different expressions that represent semantically equal programs must produce the same result. By taking expressions to expressions directly the eval operation is not so constrained.

As an example, by using lift you can write the function sum defined by

$$\begin{aligned}&\mathsf{letrec}\ \mathit{sum} = \lambda n.\\ &\qquad \mathsf{if}\ n = 0\ \mathsf{then}\ \langle\!\langle 0 \rangle\!\rangle\ \mathsf{else}\ \langle\!\langle \hat{}(\mathsf{lift}\cdot n) + \hat{}(\mathit{sum}\cdot(n-1)) \rangle\!\rangle\end{aligned}$$

This maps n to an expression that sums all the numbers up to n. For example, $\mathsf{sum}\cdot 4$ produces $\langle\!\langle 4 + 3 + 2 + 1 + 0 \rangle\!\rangle$.

This feature addresses a shortcoming of the previous version of Forte based on *FL* . Users of this system sometimes want to verify a result by case analysis that can involve decomposing a goal into hundreds of similar cases, each of which is within reach of an automatic solver. It is difficult in *FL* to write a function that will produce (a conjunction of) all those cases. Facilities like lift make this easier, and the code that does it more transparent.

8. RELATED WORK

The *reFLect* language can been seen as an application-specific contribution to the field of *meta-programming*. In Sheard's

[4]Note that of the two, it is value rather than eval that most closely corresponds to the `eval` operation in LISP.

[5]Of course, the equality of such names is not decidable.

taxonomy of meta-programming [27], *reFLect* is a framework for both generating and analyzing programs; it includes features for run-time program generation; and it is typed, 'manually staged', and 'homogeneous'. Our design decisions, however, were driven by the needs of our target applications: symbolic reasoning in higher-order logic, hardware modeling, and hardware transformation. So the 'analysis' aspect is much more important than for the design of functional meta-programming languages aimed at optimized program execution.

Nonetheless, *reFLect* has a family resemblance to languages for run-time code generation such as MetaML [33] and Template Haskell [28]. A distinguishing feature of MetaML is *cross-stage persistence*, in which a variable binding applies across the quotation boundary. The motivation is to allow programmers to take advantage of bindings made in one stage at all future stages. In *reFLect*, however, we wish to define a *logic* on top of the language and so we take the conventional logical view of quotation and binding. Variable bindings do not persist across levels. Constant definitions, however, are available in all levels. They therefore provide a limited and safe form of 'cross-level' persistence, just as they do with polymorphism.

For reasons already given, *reFLect* also differs from MetaML in typing all quotations with a universal type *term*. Template Haskell is similar to *reFLect* in this respect. One of the 'advertised goals' of Template Haskell is also to support user-defined code manipulation or optimization, though probably not logic.

Perhaps the closest framework to *reFLect* is the system described by Shields et al. in [29]. This has a universal term type, a splicing rule for quotation and antiquotation similar to our ψ rule, and run-time type checking of quoted regions. Our applications in theorem proving and design transformation have, however, led to some key differences. We adopt a simpler notion of type-consistency when splicing expressions into a context, ensuring only that the resulting expression is well typed, while the Shields system ensures consistent typing of variables. This relaxation keeps the logic we construct from *reFLect* simple, and the implementation of time critical theorem proving algorithms, like rewriting, efficient.

The *reFLect* language extends the notion of quotation and antiquotation, which have been used for term construction since the LCF system [9], by also allowing these constructs to be used for term decomposition via pattern matching. In this respect we follow the work of Aasa, Petersson and Synek [2] who proposed this mechanism for constructing and destructing object-language expressions within a meta-language. The other reflective languages discussed here [28, 29, 33] do not support this form of pattern matching, which is valuable for our applications in code inspection and transformation, but would find less application in the applications targeted by these systems.

9. CONCLUSION

In this paper we presented the language *reFLect*; a functional language with strong typing, quotation and antiquotation features for meta-programming, and reflection. The quotation and antiquotation features can be used not only to construct expressions, but also to transparently implement functions that inspect or traverse expressions via pattern matching. We made novel use of contexts with a level consistency property to give concise descriptions of the type system and operational semantics of *reFLect*, as well as using them to describe a method of compiling away the new syntactic features of *reFLect*.

We have completed an implementation of *reFLect* using the compilation technique described to translate *reFLect* into λ-calculus, which is then evaluated using essentially the same combinator compiler and run-time system as the previous *FL* system [1]. The performance of *FL* programs that do not use the new features of *reFLect* has not been impacted.

We have used *reFLect* to implement a mechanized reasoning system based on inspirations from HOL [11] and the Forte [1, 16] system, a tool used extensively within Intel for hardware verification. The ability to pattern match on expressions has made the logical kernel of this system more transparent and compact than those of similar systems. The system includes evaluation as a deduction rule, and combines evaluation with rewriting to simplify closed subexpressions efficiently.

This presentation of the type system and operational semantics for *reFLect* gives a good starting point for investigation of more theoretical properties of the language, like confluence, subject-reduction, and normalization. Sava Krstić and John Matthews of the Oregon Graduate Institute have proved these properties for the *reFLect* language features for expression construction and analysis, though not those that relate to evaluation of expressions [20]. Their proofs cover the language presented here up to, but not including, section 7.

10. ACKNOWLEDGMENTS

The authors would like to thank the following people for their helpful discussions on *reFLect*: Carl Seger of Intel Corporation, Mike Gordon of the University of Cambridge, Sava Krstić and John Matthews of the Oregon Graduate Institute, and Mark Shields of Galois Connections.

11. REFERENCES

[1] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. *Lifted-FL*: A pragmatic implementation of combined model checking and theorem proving. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *LNCS*, pages 323–340. Springer-Verlag, 1999.

[2] A. Aasa, K. Petersson, and D. Synek. Concrete syntax for data objects. In *LISP and Functional Programming: ACM Conference, LFP 88*, pages 96–105. ACM Press, 1988.

[3] B. Barras. Proving and computing in hol. In *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *LNCS*, pages 17–37. Springer-Verlag, 2000.

[4] Z. E.-A. Benaissa, E. Moggi, W. Taha, and T. Sheard. Logical modalities and multi-stage programming. In *Intuitionsitic Modal Logics and Applications: Federated Logic Conference Satellite Workshop, IMLA*, 1999.

[5] S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs: International Workshop, TYPES 2000*, volume 2277 of *LNCS*, pages 24–40. Springer-Verlag, 2000.

[6] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Functional Programming: International Conference, ICFP'98*, pages 174–184. ACM Press, 1998.

[7] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[8] T. Coquand. An analysis of Girard's paradox. In *Logic in Computer Science: 1st IEEE Symposium, LICS'86*, pages 227–236. IEEE Computer Society Press, 1986.

[9] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer-Verlag, 1979.

[10] M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: Workshop*, pages 153–177. North-Holland, 1985.

[11] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[12] J. Grundy, T. Melham, and J. O'Leary. A reflective functional language for hardware design and theorem proving. Technical Report PRG-RR-03-16, Programming Research Group, Oxford University Computing Laboratory, October 2003.

[13] R. Harper, D. MacQueen, and R. Milner. Standard ML. Report 86-2, University of Edinburgh, Laboratory for Foundations of Computer Science, 1986.

[14] J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, 1995.

[15] S. D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. MIT Press, 1984.

[16] R. B. Jones, J. W. O'Leary, C.-J. H. Seger, M. D. Aagaard, and T. F. Melham. Practical formal verification in microprocessor design. *IEEE Design & Test of Computers*, 18(4):16–25, 2001.

[17] R. Kaivola and K. R. Kohatsu. Proof engineering in the large: Formal verification of the Pentium® 4 floating-point divider. In T. Margaria and T. F. Melham, editors, *Correct Hardware Design and Verification Methods: 11th Advanced Research Working Conference, CHARME 2001*, volume 2144 of *LNCS*, pages 196–211. Springer-Verlag, 2001.

[18] R. Kaivola and N. Narasimhan. Formal verification of the Pentium® 4 multiplier. In *High-Level Design Validation and Test: 6th International Workshop: HLDVT 2001*, pages 115–122, 2001.

[19] M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer, 2000.

[20] S. Krstić and J. Matthews. Subject reduction and confluence for the *reFLect* language. Technical Report CSE-03-014, Department of Computer Science and Engineering, OGI School of Science and Engineering at Oregon Health and Sciences University, 2003.

[21] A. C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. Macdonald, 1969.

[22] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *Computer Languages: International Conference*, pages 90–101. IEEE Computer Society Press, 1998.

[23] T. F. Melham. *Higher Order Logic and Hardware Verification*. Cambridge University Press, 1993.

[24] O. Müller and K. Slind. Treating partiality in a logic of total functions. *The Computer Journal*, 40(10):640–651, 1997.

[25] L. C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3(2):119–149, 1983.

[26] E. Pašalić, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. *SIGPLAN Notices*, 37(9):218–229, 2002.

[27] T. Sheard. Accomplishments and research challenges in meta-programming. In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation: 2nd International Workshop, SAIG 2001*, volume 2196 of *LNCS*, pages 2–44. Springer-Verlag, 2001.

[28] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Haskell: Workshop*, pages 1–16. ACM Press, 2002.

[29] M. Shields, T. Sheard, and S. P. Jones. Dynamic typing as staged type inference. In *Principles of Programming Language: 25th Annual Symposium, POPL 1998*, pages 289–302, 1998.

[30] G. Spirakis. Leading-edge and future design challenges: Is the classical EDA ready? In *Design Automation: 40th ACM/IEEE Conference, DAC 2003*, page 416. ACM Press, 2003.

[31] P. C. Suppes. *Introduction to Logic*, chapter 6. Van Nostrand Reinhold, 1957.

[32] W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[33] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. *SIGPLAN Notices*, 32(12):203–217, 2002.

Verifying the ARM Block Data Transfer Instructions

Anthony Fox
Computer Laboratory, University of Cambridge

Abstract

The HOL-4 proof system has been used to formally verify the correctness of the ARM6 *micro-architecture*. This paper describes the specification and verification of one instructions class, *block data transfers*; these are a form of load-store instruction in which a set of up to sixteen registers can be transferred atomically. The ARM6 is a commercial RISC microprocessor that has been used extensively in embedded systems – it has a 3-stage pipeline with a multi-cycled execute stage. A list based *programmer's model* specification of the block data transfers is compared with the ARM6's implementation which uses a 16-bit mask. The models are far removed and reasonably complex, and this poses a verification challenge. This paper describes the approach and some key lemmas used in verifying correctness, which is defined using data and temporal abstraction maps.

1 Introduction

This paper presents a HOL specification of the ARM block data transfer instruction class [7, 18], together with a description of the ARM6 implementation and its formal verification using the HOL proof system. This work builds upon an ARM6 verification [6] which did not cover the block data transfer or multiply instruction classes.

The correctness model and underlying approach used for this work has been formalised in HOL [5]. This methodology was developed at Swansea and work has continued there using Maude [9, 10]. Using this approach, the correctness of the ARM6 implementation of the block data transfers has been formally verified. This is achieved by relating state machine models at the instruction set and micro-architecture levels of abstraction.

One source of difficultly in verifying this instruction class is the relatively complex nature of the implementation. The ARM6 has a 3-stage pipeline with fetch, decode and execute stages, and the execute stage can take a number of processor clock cycles to complete. With most instructions the number of cycles required is a small constant value. For example, an ordinary (single) load instruction takes three cycles, or five if the program counter is modified. The processor control logic makes use of a counter (the *instruction sequence*, `is`) to implement this behaviour. Typically this counter is incremented after each execute cycle and this provides a simple mechanism to symbolically execute an instruction to completion. However, with block data transfer instructions the counter takes and holds the value `tn` until a termination condition is met (this can take up to sixteen cycles). A 16-bit mask is used to keep track of which registers have been transfered and this forms the basis for the termination test. Consequently, symbolic execution for this instruction class is not straightforward.

From a correctness standpoint one must consider the case of writing to the memory at the address $pc + 8$ or $pc + 4$, where pc is the address of the instruction being executed. These addresses correspond with instructions that have been fetched and decoded respectively.[1] In order to provide a clean model, the ARM6 *should* detect when these instructions have been updated by a memory write and take steps to fetch and decode them again. However, the ARM6 does not waist costly control logic in dealing with this, instead it just carries on regardless. Before the block data transfers were verified two solutions to this problem were applied (for the verification of single word/byte data stores):

1. No-clobber method: a write to the addresses $pc + 8$ and $pc + 4$ is nullified at the programmer's model and micro-architecture levels.

[1] The architecture does not split the main memory into program and data parts. Memory is byte addressable and each 32-bit instruction occupies four bytes.

User	FIQ	IRQ	SVC	Abort	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13	r13_fiq	r13_irq	r13_svc	r13_abt	r13_und
r14	r14_fiq	r14_irq	r14_svc	r14_abt	r14_und
r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_irq	SPSR_svc	SPSR_abt	SPSR_und

Figure 1: ARM's visible registers.

2. Data forwarding implementation: a write to these addresses is detected by the processor and the fetch and decode components of the state are updated as appropriate.

The second method has a clean programmer's model specification but it complicates, and does not accurately model, the micro-architecture. Both of these methods were manageable with `STR` instructions but they would have significantly added to the difficulty of verifying the block data transfers. Therefore, a third method has been adopted: the programmer's model is augmented with two registers forming a rudimentary pipeline. A further more abstract level of abstraction has been introduced to hide the pipeline i.e. the model accepts a stream of instructions, abstracting out details about instruction fetching.

1.1 Related Work

Early work on the mechanical verification of processors includes: TAMARACK [13], SECD [8], the partial verification of Viper [4], Hunt's FM8501 [11], and the generic interpreter approach of Windley [19]. Following this work, Miller and Srivas verified some of the instructions of a simple commercial processor called the AAMP5 [16]. Complex commercial designs have also been specified, simulated and verified using ACL2 [2, 14].

With the addition of complex multi-stage pipelines and out-of-order execution, contemporary commercial designs were considered too complex for *complete* formal verification. Recently progress has been made in verifying academic designs based around Tomasulo's algorithm [15, 12, 17, 1]. The instruction sets used for this work are often relatively simple (i.e. no block data transfers) with many based on the DLX architecture of Hennessy and Patterson. It is worth noting that, in this later work, the correctness models are based on *incremental flushing* [3]. This means that the processor model is only guaranteed to correspond with the instruction set model provided that the processor's pipeline is flushed after each instruction execution. This will catch many potential bugs, but a stronger correctness model is used here – the processor's state is guaranteed to be correct even in the absence of flushing.

2 The Instruction Set Architecture

For details of the ARM programmer's model the reader is referred to Furber [7] and the ARM Architecture Reference manual [18]. A limited précis is provided here.

The ARM architecture's visible state consists of a main memory and a set of 32-bit registers. The main memory is effectively an array of 2^{32} bytes. The registers form overlapping banks, as shown in Figure 1. Six *processor modes* provide support for exception handling and system-level programming. The general purpose registers are named `r0` to `r14`, the program counter is `r15`, and CPSR is the Current Program Status Register. When not in user mode the programmer also has access to a Saved Program Status Register (SPSR). The CPSR stores the current processor mode, together with four flags: `N` (negative), `Z` (zero), `C` (carry) and `V` (overflow). These flags are used to control program flow: all instructions are conditionally executed. For example, the instruction `STMHI` will be a no-op if `C` is clear or `Z` is set.

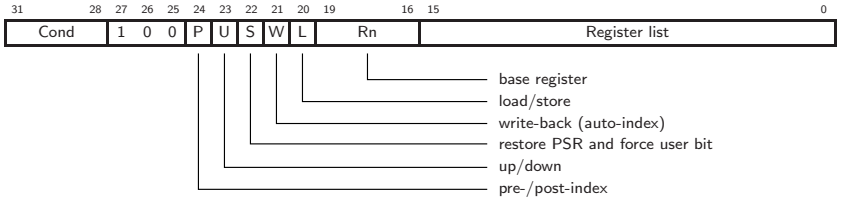


Figure 2: Encoding for the block data transfer instruction class.

2.1 Block Data Transfers

Block data transfer instructions load/store a set of general purpose register values from/to main memory; the instruction format is shown in Figure 2. These instructions are used for procedure entry and return (saving and restoring workspace registers), and in writing memory block copy routines.

The set of registers to be transferred is encoded using a 16-bit value; the program counter may be included in the list (bit fifteen). The memory block is determined by the base register `Rn`, and the bits `P` and `U`. The `W` flag enables base register write back (auto-indexing). There are also special forms of the instruction for accessing the user mode registers (when not in user mode) and for restoring the CPSR when returning from an exception – these options are controlled by the `S` flag and bit fifteen.

The instruction syntax is illustrated below:

```
LDM|STM{<cond>}<add mode> Rn{!}, <registers>
LDM{<cond>}<add mode> Rn{!}, <registers + pc>^
LDM|STM{<cond>}<add mode> Rn, <registers - pc>^
```

Here `<cond>` is a condition code, `<addr mode>` is the address mode and `<registers>` is a list of registers. The block copying address modes are `IA`, `IB`, `DA` and `DB` – as indicated these increment/decrement the address register after/before each memory access.[2] An `!` is used for base register write-back, and the suffix `^` is used to set the `S` flag.

As an example, if the processor is in supervisor mode with the `Z` flag set, the instruction

```
LDMEQDB r0!, {r1,r2,pc}^
```

will perform the following assignments:

```
r0 ← r0-12; r1 ← mem[r0-12]; r2 ← mem[r0-8]; r15 ← mem[r0-4]; CPSR ← SPSR_svc .
```

All transfers are ordered: registers with lower indices are mapped to lower memory addresses.

The register list should not be empty i.e. the lowest sixteen bits of the op-code should not all be clear. This restriction will be enforced by any sensible compiler and/or assembler, but this does not guarantee that such instructions can never be executed (it is trivial to write an assembly program that generates and then executes such an instruction). The ARM6 has an unfortunate load multiple behaviour when the register list is empty – a load to the program counter occurs. Rather than specify this at the programmer's model level, the HOL model of the ARM6 has been modified to give a more sensible behaviour i.e. no load occurs.

With block stores, if the program counter is in the list then the value stored is implementation dependent. If the base register is in the list then write-back should not be specified because the result is *unpredictable*. The HOL programmer's model specification has been tailored to conform with ARM6 behaviour for these cases.

2.2 A HOL Specification

The HOL specification of the block data transfers is shown in Figure 3. The function `LDM_STM` takes the current programmer's model state, the processor mode and the instruction op-code, and it gives the next state. This function is only called when it is established that op-code `n`

[2]Stack based mnemonics are available as an alternative: `FA`, `FD`, `EA` and `ED` are used to implement full/empty, ascending/descending stacks.

```
⊢def LDM_STM (ARM mem reg psr) mode n =
      let (P,U,S,W,L,Rn,pc_in_list) = DECODE_LDM_STM n in
      let rn = REG_READ reg mode Rn in
      let (bl_list,rn') = ADDR_MODE4 P U rn n
      and mode' = if S ∧ (L ⇒ ¬pc_in_list) then usr else mode
      and pc_reg = INC_PC reg in
      let wb_reg =
           if W ∧ ¬(Rn = 15) then REG_WRITE pc_reg (if L then mode else mode') Rn rn' else pc_reg
      in
         if L then
           ARM mem (LDM_LIST mem wb_reg mode' bl_list)
             (if S ∧ pc_in_list then CPSR_WRITE psr (SPSR_READ psr mode) else psr)
         else
           ARM (STM_LIST mem (if FST (HD bl_list) = Rn then pc_reg else wb_reg) mode' bl_list) wb_reg psr
```

Figure 3: Programmer's model specification of block data transfers.

is a block data transfer instruction that passes the conditional execution test. The state space constructor `ARM` takes a triple: the memory `mem`, the general purpose registers `reg`, and the program status registers `psr`.

Although the definition of `LDM_STM` is not especially large, there are some subtle aspects to the semantics of block data transfers. Depending on the context, the processor mode is either `mode` or `mode'` (which might be set to user mode), therefore one must pay attention as to which mode is being used when accessing registers. The register bank after incrementing the program counter is denoted by `pc_reg` and after register write-back this becomes `wb_reg`. If the *first* register of a block store is the base register then the value `rn` is stored (i.e. `pc_reg` is used), otherwise write-back may have occurred and `rn'` is stored (`wb_reg` is used). Write-back occurs only if the base register is not the program counter.

There are four key sub-functions: `DECODE_LDM_STM`, `ADDR_MODE4`, `LDM_LIST` and `STM_LIST`; these are defined in Appendix B. The function `DECODE_LDM_STM` takes the op-code and splits it into seven fields. For example, the instruction `LDMEQDB r0!, {r1,r2,pc}^` is encoded with the natural number 158367750, and this decodes as follows:

```
⊢ DECODE_LDM_STM 158367750 = (T,F,T,T,T,0,T) .
```

The function `ADDR_MODE4` takes the address mode flags (`P` and `U`), the base address `rn` and the op-code `n`, and it gives a pair `(bl_list,rn')`. With our example:

```
⊢ ADDR_MODE4 T F rn 158367750 = ([(1,rn - 0xC); (2,rn - 0x8); (15,rn - 0x4)],rn - 0xC) .
```

If write-back is enabled then register `Rn` takes the value `rn'`; `bl_list` consists of pairs of the form `(rp,addr)` where `rp` is a register index and `addr` is the corresponding memory address. A function `REGISTER_LIST` gives the list of register indices and this is 'zipped' with the memory block addresses.

The function `LDM_LIST` folds the list `bl_list` with a memory-read, register-write operation to give the next state of the register bank. Likewise, the function `STM_LIST` folds the list with a register-read, memory-write operation to give the state of the main memory.

This list based specification is compact and hopefully clear. Consequently, one can be confident that the specification is consistency with respect to the reference [18] – it also provides a model that can be executed efficiently. However, the verification must bridge a large gap between this abstract semantics and the concrete processor implementation.

3 The Micro-architecture

A simplified view of the ARM6 data path is shown in Figure 4; the components of the data path (busses, latches, multiplexers and functional units) are used in executing all of the ARM instructions. When reading from memory the data is transfered to the data-in register `din`. When writing to memory the data is placed on the B bus. The address register `areg` may be updated using the program counter, the address incrementer, or output from the ALU.

Block data transfers are multi-cycled instructions; their execution can take from two to twenty cycles to complete. The execute stage is split into sub-stages and these are shown

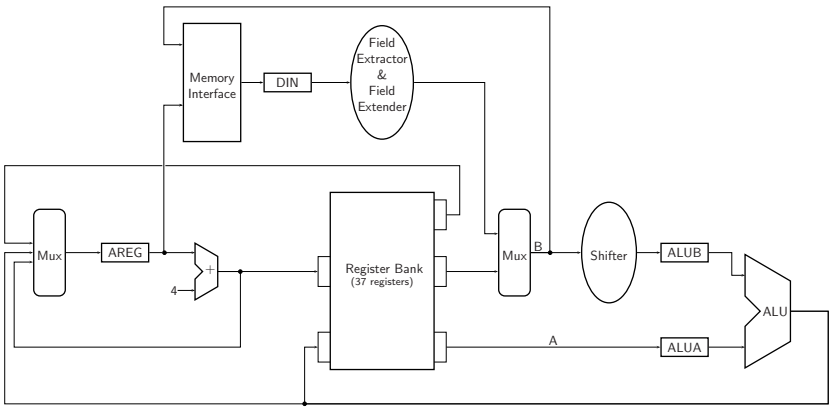


Figure 4: The ARM6 Data Path.

in Tables 1(a) and 1(b) for the block load and store instructions respectively. The first two cycles are needed for address computation and base register write-back, and then sub-stage t_n is repeated $\ell - 1$ times, where ℓ is the length of the register list. The leftmost columns in the tables show the value of the instruction sequence counter (`is`) – this component forms part of the processor's control logic. Note that the cycle t_2 is really the last execute cycle of the previous instruction and so in this case the component `is` does not actually take this value.

The first memory address (start of the memory block) is computed at cycle t_3 using an offset; this value is then stored in the address register. With `DB` addressing the offset is $4 * (\ell - 1) + 3$ and the first address is $\neg \mathit{offset} + rn$, where rn is the value of base register and $\neg x$ is the 32-bit one's complement of x. In our example (Section 2.1), the offset is 11 and the first address is $rn - 12$. On successive cycles the address register is incremented by four (only word access is supported). The last (write-back) address is computed at cycle t_4. With the `DB` address mode the write-back address is the same as the first address (i.e. $rn - 12$), but this is not the case for all of the other addressing modes.

A 16-bit mask is used to compute the register index used for each data transfer. The component `rp` stores the index for the next register to be processed – this is the *priority* register. The computation of `rp` (with two time shifted copies: `orp` and `oorp`) is shown below for our example instruction (op-code 158367750):

`is`	`mask`	`mask` $\wedge_{16}$ `ireg`	`rp`	`orp`	`oorp`
t_3	1111111111111111	1000000000000110	1		
t_4	1111111111111101	1000000000000100	2	1	
t_n	1111111111111001	1000000000000000	15	2	1
t_n	0111111111111001	0000000000000000	$\perp$	15	2
t_m	$\perp$	$\perp$	$\perp$	$\perp$	15

The operation $\wedge_{16}$ represents bitwise conjunction for 16-bit values. The t_n cycle lasts for two cycles – it is repeated until the mask conjunction (column three) is zero. The data transfers always occur in ascending register index order, irrespective of the addressing mode. Consequently, the priority register is always the lowest index position for a set bit in the conjunction. If the conjunction is zero then the value of `rp` is undefined and, consequently, so are the derived values; undefined values are represented by the symbol $\perp$.

In actuality, if `ireg[15:0]` is zero in a block load then an ARM6 will carry out a load with register fifteen as the destination register and this will instigate a branch. This is because of the way in which the index search is implemented in hardware (i.e. it gives the last possible value, which is fifteen) and the fact that the t_m cycle always occurs with block loads. It was decided not to model this counter intuitive behaviour in the HOL specification – instead, the value of `rp` is undefined and the control logic has been modified to prevent the load going ahead in this case – this gives a cleaner programmer's model specification. The block stores did not need modifying because the processor naturally avoids storing a value when the list is empty.

In our example there is a load to register fifteen and so a branch will occur. This means that additional cycles are required for instruction fetch and decode; the instruction will actually

Table 1: The sub-stages in the ARM6's block transfer execution. Cycle t_n is repeated until the bitwise conjunction of the mask and `ireg[15:0]` is zero.

(a) Block loads.

t_2	Set `mask` to `0xFFFF`
t_3	Fetch an instruction Increment the program counter Set `areg` to the first address (using `reg[Rn]` and `offset`) Set `rp` using `ireg` and `mask` Clear `rp` bit of `mask` Set `orp` to be `rp`
t_4	If write-back enabled then set reg[Rn] to the last address (using `offset`) Load `din` with `mem[areg]` Increment `areg` Set `rp` using `ireg` and `mask` Clear `rp` bit of `mask` Set `oorp` to be `orp` Set `orp` to be `rp`
t_n	Set `reg[oorp]` to be `din` Load `din` with `mem[areg]` Increment `areg` Set `rp` using `ireg` and `mask` Clear `rp` bit of `mask` Set `oorp` to be `orp` Set `orp` to be `rp`
t_m	Set `reg[oorp]` to be `din` Set `areg` to be program counter value If bit twenty-two and bit fifiteen of `ireg` set then set `cpsr` to be `spsr` Decode the next instruction

(b) Block stores.

t_2	Set `mask` to `0xFFFF`
t_3	Fetch an instruction Increment the program counter Set `areg` to the first address (using `reg[Rn]` and `offset`) Set `rp` using `ireg` and `mask` Clear `rp` bit of `mask` Set `orp` to be `rp`
t_4	If write-back enabled then set reg[Rn] to the last address (using `offset`) Store `reg[orp]` to `mem[areg]` Increment `areg` Set `rp` using `ireg` and `mask` Clear `rp` bit of `mask` Set `orp` to be `rp`
t_n	Store `reg[orp]` to `mem[areg]` If the last cycle then set `areg` to the program counter value and decode the next instruction otherwise increment `areg` Set `rp` using `ireg` and `mask` Clear `rp` bit of `mask` Set `orp` to be `rp`

fully complete two cycles after the t_m cycle.

3.1 A HOL Specification

A HOL specification of the ARM6 without block data transfers [6] was extended to cover this instruction class – most modifications to the previous specification are obvious and do not merit documenting here. Appendix C presents functions that were new to the specification.

The function `NBS` specifies the mode change caused by the `S` flag option i.e. it determines when the user mode is activated. The mask behaviour (as described in the previous section) is implemented using the functions `MASK`, `RP` and `PENCZ`. The function `MASK` defines the state of the mask – if the next instruction class (`nxtic`) is a block transfer then the initial mask value is `0xFFFF` and on subsequent cycles a single mask bit is cleared using the function `CLEARBIT`, see Appendix A. The masking is modelled using natural numbers – this is done simply to avoid introducing additional operator overloading i.e. by simultaneously loading 16-bit and 32-bit words theories.

The priority register `rp` is given by the function `RP`; this computes the bitwise conjunction of the register list and the mask, then the lowest set bit is determined using the function

$$\vdash_{def} \texttt{LEASTBIT n = LEAST b. BIT b n .}$$

When the register list is exhausted `rp` takes the value `LEASTBIT 0`, which is undefined; in HOL this is an unspecified natural number value. The predicate `PENCZ` holds true only when this termination condition is met. The function `LEASTBIT` is not readily executable (i.e. one cannot evaluate ground terms by adding the definition to a HOL *compset*); this is because

the LEAST operation introduces non-termination problems. When simulating the ARM6, a nested-if expansion of LEASTBIT (for all $b < 16$) is used.

The function OFFSET is used to compute the first and write-back addresses on cycles t_3 and t_4. The number of registers in the register list is determined using SUM and BITV (Appendix A). Thus, the number of registers is $\sum_{i=0}^{15} \text{bitv}(ireg)(i)$. Depending on the addressing mode (and the instruction sequence) the offset, its one's complement, or zero is added to the base register address; this is implemented by the processor's ALU.

4 Correctness

The correctness model used for the ARM6 verification has been formalised in HOL [5]. The following sections introduce the models and abstractions that are used in establishing the ARM6 processor's correctness.

Store instructions require special attention because they can invalidating the state of a processor's pipeline [6]. This problem is heightened by the inclusion of block data transfers. For example, consider the following fragment of ARM code:

```
       ADR   r0, label
       STMIA r0, {r1,r2}
label: MOV   r3, #10
       MOV   r4, #12
```

The first instructions sets register r0 to the label address. The STMIA instruction then stores registers r1 and r2 to this and the following address, thus overwriting the two MOV instructions. However, rather than execute the new instructions (i.e. r1 and r2), an ARM6 will actually execute both of the MOV instructions. These instructions are preserved because they have entered the *pipeline* and the processor only *flushes* the instruction pipeline after a branch (write to the program counter). Of course, this example has been construed so as expose the pipeline and hence be unsafe. In practice, it is not worth wasting valuable processor logic on handling such fundamentally flawed code, and this is the position that was adopted by the designers of the ARM6. However, from a correctness standpoint, this must be dealt with. The approach adopted here is to augment the programmer's model state space with two 32-bit registers and these hold the op-codes of the fetched and decoded instructions. By implementing a rudimentary pipeline at the ISA level the correspondence between our models is easier to establish. The ISA pipeline is really just a buffer: the model still occupies the same level of temporal abstraction i.e. each cycle always corresponds with the execution of a single instruction. One criticism that could be made of this approach is that the semantics of the abstract model is now too concrete when compared with the reference description [18]. On the other hand, our model is a verified abstraction of the ARM6 and so one can be wholly confident that it faithfully simulates the processor regardless of the code being executed. However, it is not a suitable target for other ARM processors because there will be differences with respect to the *unpredictable* parts of the programmer's model. In order to unite the ARM processor family, one must introduce another level of abstraction and construct a non-deterministic instruction set model.

4.1 State Functions

The ARM architecture and ARM6 processor are modelled in HOL with the following functions:

```
STATE_ARM_PIPE:num→state_arm_pipe→state_arm_pipe
STATE_ARM6:num→state_arm6→state_arm6
```

A constructor ARM_PIPE extends the programmer's model state space (state_arm) with the state of the pipeline. The two additional 32-bit words are named ireg and pipe – they form a simple buffer which is emptied and re-filled when a branch occurs. This enables the ISA model to simulate ARM6 behaviour when storing data to the addresses $pc + 4$ and $pc + 8$.

With the inclusion of the block data transfers, the processor's state space (state_arm6) now contains three additional components: mask, orp and oorp. For convenience these components are of type num, but they are more naturally 16-bit and 4-bit values.

```
⊢def DUR_ARM6 (ARM6 mem (DP reg psr areg din alua alub) (CTRL pipea pipeaval ..  mshift)) =
    let (nzcv,m) = DECODE_PSR (CPSR_READ psr) in
    let abortinst = ABORTINST iregval onewinst ointstart ireg nzcv in
    let ic = IC abortinst nxtic in
    let len = LENGTH (REGISTER_LIST (w2n ireg))
    in
       if ic = undef then
         4
       else if ..
         ..
       else if ic = ldm then
         2 + (len - 1) + 1 + (if WORD_BIT 15 ireg then 2 else 0)
       else if ic = stm then
         2 + (len - 1)
       else if ..
        ..
```

Figure 5: The duration map DUR_ARM6.

4.2 Data and Temporal Abstraction

A data abstraction ABS_ARM6:state_arm6→state_arm_pipe is defined as follows:

```
⊢def ABS_ARM6 (ARM6 mem (DP reg psr areg din alua alub)
       (CTRL pipea pipeaval pipeb pipebval ireg iregval ointstart
          onewinst opipebll nxtic nxtis aregn nbw nrw sctrlreg psrfb
            oareg mask orp oorp mul mul2 borrow2 mshift)) =
       ARM_PIPE (ARM mem (SUB8_PC reg) psr) pipeb ireg
```

The state components are grouped into vectors using five constructors: ARM6, DP (the data path), CTRL (the processor control), ARM_PIPE and ARM. The data abstraction projects out the pipeline state (pipeb and ireg) and the visible state components (mem, reg and psr). The function SUB8_PC is used to subtract eight from the ARM6's program counter value, which is eight bytes ahead of the address of the instruction being executed.

A *uniform immersion* [5] specifies the temporal relationship between the cycles of the ARM6 processor and single instruction execution. A function DUR_ARM6:state_arm6→num specifies the number of cycles required to complete the execution of an instruction. A fragment of this function, giving the timings for block data transfers, is shown in Figure 5. The instruction class (ic) and the length of the register list (len) are used to determine how long the block data transfer will take. The timings are presented as sums; this splits the execution into distinct phases. For example, with an LDM instruction the first two cycles are t3 and t4, then there are $\ell - 1$ cycles of tn, followed by one cycle of tm, and finally two extra cycles if the program counter is in the register list.

This duration function is only valid for processor states in which the pipeline is full i.e. the first execute cycle is about to commence – the component nxtis must have the value t3. An initialisation function for the ARM6 is provided in the following section. During verification one must show that the timings specified above are consistent with passing from one initial state to another.

4.3 Initialisation

An initialisation function INIT_ARM6:state_arm6→state_arm6 is used to ensure that the processor starts in a valid state. This function takes a state and converts it into an initial version – it is an identity mapping on valid initial states:

```
⊢def INIT_ARM6 (ARM6 mem (DP reg psr areg din alua alub) (CTRL pipea pipeaval .. mshift)) =
    let nxtic' = DECODE_INST (w2n ireg) in
      ARM6 mem (DP reg psr (REG_READ6 reg usr 15) ireg alua alub)
        (CTRL pipea T pipeb T ireg T F T T nxtic' t3 2 nbw F sctrlreg
           psrfb oareg (MASK nxtic' t3 mask ARB) orp oorp mul mul2 borrow2 mshift)
```

This function differs significantly from the earlier verification [6] – our ISA model now has a pipeline and so the pipeline components (pipea, pipeb and ireg) can be initialised with any values. If these values are not consistent with the instructions in memory (corresponding

with the current value of the program counter) then conceivably this could be because a store instruction has just invalidated the pipeline's state. However, this is not a problem because the pipeline state is visible at the ISA level, via the data abstraction `ABS_ARM6`. The components `orp` and `oorp` are not altered, but the mask is set using the function `MASK` – if the next instruction class is a block data transfer then this will set `mask` to the value `0xFFFF`.

With respect to initialisation, there are three classes of component:

- The visible state components: `mem`, `reg`, `psr`, `pipeb` and `ireg`. These components cannot be altered during initialisation because otherwise correctness would fail at time zero.
- State components whose initial values are of significance. For example, the next instruction class (`nxtic`) must be the decoding of the instruction register (`ireg`).
- State components whose initial values are of no significance. For example, the ALU registers (`alua` and `alub`) can take any values initially.

As a general rule an initialisation function should be as weak as possible i.e. it should only alter state components that are of the second type. In this context the initialisation represents an invariant for the design. However, the initialisation function is actually viewed as *part* of the design i.e. it is used to define the state function `STATE_ARM6` and is used when simulating the processor.

4.4 Correctness Definition

The ARM6 is considered correct if:

Commutativity Theorem

```
⊢ ∀t a. STATE_ARM_PIPE t (ABS_ARM6 a) = ABS_ARM6 (STATE_ARM6 (IMM_ARM6 a t) a)
```

where `IMM_ARM6` is the uniform immersion with duration function `DUR_ARM6`. To ensure that the implementation covers all of the specification's behaviour, one must also show that the data abstraction is a surjective mapping i.e. each initial specification state must have at least one initial implementation state that maps to it. This condition is relatively easy to verify for `ABS_ARM6` because the operation `SUB8_PC` has an obvious inverse. The main focus of the formal verification is, therefore, the commutativity theorem.

5 Formal Verification

The correctness condition presented in Section 4.4 is universally quantified over time (the natural numbers). Using the one-step theorems [5], it is sufficient to prove that the following four theorems hold:

```
1  ⊢ ∀a. (STATE_ARM6 (IMM_ARM6 a 0) a = a') ⇒ (INIT_ARM6 a' = a')
2  ⊢ ∀a. (STATE_ARM6 (IMM_ARM6 a 1) a = a') ⇒ (INIT_ARM6 a' = a')
3  ⊢ ∀a. STATE_ARM_PIPE 0 (ABS_ARM6 a) = ABS_ARM6 (STATE_ARM6 (IMM_ARM6 a 0) a)
4  ⊢ ∀a. STATE_ARM_PIPE 1 (ABS_ARM6 a) = ABS_ARM6 (STATE_ARM6 (IMM_ARM6 a 1) a)
```

The first and third theorems are trivial; the main verification effort lies in verifying the second and forth theorems. Here, the next state function `NEXT_ARM6` is iterated using `FUNPOW`, with the number of iterations given by the map `DUR_ARM6 ∘ INIT_ARM6`. The proof proceeds with case splitting over the instruction class and this normally gives a small constant value for the number of iterations. However, with the block data transfer instruction class, the duration is a function of the length of the register list. Exhaustive proof over all of the 2^{16} possible register lists is not a viable option, especially considering that further case splitting is required for each combination of addressing mode and the options `S`, `W`, `L`, `Rn = 15` and `pc_in_list`.

5.1 Approach

In order to verify the block data transfers, invariants are constructed for the iterated t_n-phase of the execution. This phase occurs two cycles into the execution and accounts for $\ell - 1$ cycles, where ℓ is the length of the register list. Three cases must be considered: $\ell = 0$, $\ell = 1$ and

$1 < \ell$. In the first two cases the t_n-phase does not occur; with stores this means that the execution is complete after the t_3 and t_4 cycles, but with loads completion occurs after the t_m cycle, which will be followed by two extra cycles if the list is `{r15}`. Therefore, invariants are only needed when $1 < \ell$.

Let A be the processor's state space and $f : A \to A$ be the next state function. The state space has two disjoint subsets $X_c = \text{Image}(f^2, I_c)$, where I_c is the set of initial states for the classes $c \in \{\texttt{ldm}, \texttt{stm}\}$. Induction is used to verify that the functions $g_c : \mathbb{N} \times X_c \to A$ have the property: for all $a \in X_c$, $1 < \ell$ and $i < \ell - 1$

$$g_c(i, a) = f^i(a) \ .$$

The functions g_c are a form of invariant; they were constructed manually – an initial definition was made (guessed at) and this was refined until the final version was proved to be valid. Functions $h_c : X_c \to A$ are defined by

$$h_c(a) = f(g_c(\ell - 2, a)) = f^{\ell-1}(a) \ .$$

At cycle $\ell - 2$ the termination condition is about to be met and so each function h_c maps states in the set X_c to states corresponding with the end of the t_n-phase of execution.

Using the functions h_c it is now possible to express the state of the processor after completing the execution of the block data transfers; for all $a \in I_c$ the final state is:

$$\begin{cases} f^2(a), & \text{if } c = \texttt{stm} \text{ and } \ell = 0, 1, \\ h_{\texttt{stm}}(f^2(a)), & \text{if } c = \texttt{stm} \text{ and } 1 < \ell, \\ f^3(a), & \text{if } c = \texttt{ldm} \text{ and } \ell = 0, \\ f^3(a), & \text{if } c = \texttt{ldm} \text{ and } \ell = 1 \text{ and } \texttt{r15} \text{ not in list}, \\ f^5(a), & \text{if } c = \texttt{ldm} \text{ and } \ell = 1 \text{ and } \texttt{r15} \text{ in list}, \\ f(h_{\texttt{ldm}}(f^2(a))), & \text{if } c = \texttt{ldm} \text{ and } 1 < \ell \text{ and } \texttt{r15} \text{ not in list}, \\ f^3(h_{\texttt{ldm}}(f^2(a))), & \text{if } c = \texttt{ldm} \text{ and } 1 < \ell \text{ and } \texttt{r15} \text{ in list}. \end{cases}$$

The initial state sets I_c are generated using the initialisation function. Having determined the state of the processor at the times given by the duration function, it is then necessary to relate these states to those of the specification. The following sections indicate how the h_c functions were constructed and show how the masking used in the processor model is related to the list model used in the ISA specification.

5.2 Lemmas about Priority Register Masking

The following functions are defined in HOL:

```
⊢def GEN_RP wl ireg mask = LEASTBIT (BITWISE wl (∧) ireg mask)
⊢def MASK_BIT wl ireg mask = CLEARBIT wl (GEN_RP wl ireg mask) mask
⊢def MASKN wl n ireg = FUNPOW (MASK_BIT wl ireg) n (ONECOMP wl 0)
```

These function generalise those of the ARM6 specification to an arbitrary mask length `wl`; this enables results to be proved by induction over the word length. The function `MASKN` gives the n^{th} value of the mask; with our block load example:

```
⊢ RP ldm (BITS 15 0 158367750) (MASKN 16 0 158367750) = 1
⊢ RP ldm (BITS 15 0 158367750) (MASKN 16 1 158367750) = 2
⊢ RP ldm (BITS 15 0 158367750) (MASKN 16 2 158367750) = 15
⊢ PENCZ (BITS 15 0 158367750) (MASKN 16 3 158367750)
```

These values correspond with those in the table on page 5.

The ISA level function `REGISTER_LIST` is also generalised to an arbitrary length:

```
⊢def GEN_REG_LIST wl a = (MAP SND o FILTER FST) (GENLIST (λb. (BIT b a,b)) wl)
```

Two key lemmas relate this function with the ARM6 model:

```
⊢ ∀wl ireg. LENGTH (GEN_REG_LIST wl ireg) = SUM wl (BITV ireg)

⊢ ∀wl ireg n. n < LENGTH (GEN_REG_LIST wl ireg) ⇒
              (EL n (GEN_REG_LIST wl ireg) = GEN_RP wl ireg (MASKN wl n ireg))
```

The first theorem shows that the length of the list is equal to the sum of the constituent bits. The second theorem shows that n^{th} element of the register list corresponds with the priority register obtained using the n^{th} mask value. Specialising `wl` to be sixteen then provides a connection between the function `REGISTER_LIST`, and the functions `RP` and `MASK`. The second lemma uses the first and it requires some work to prove: `GEN_REG_LIST` uses the primitives `MAP`, `FILTER`, `GENLIST` and `BIT`; and `MASKN` uses `FUNPOW`, `LEAST`, `BITWISE`, `ONECOMP` and `BIT`.

Another key lemma concerns the termination condition:

```
⊢ ∀ic. (ic = ldm) ∨ (ic = stm) ⇒
        (∀a n. n < LENGTH (REGISTER_LIST a) ⇒ ¬PENCZ ic a (MASKN 16 n a)) ∧
         ∀a. PENCZ ic a (MASKN 16 (LENGTH (REGISTER_LIST a)) a)
```

This lemma shows that the termination predicate `PENCZ` is false up until the ℓ^{th} mask value.

5.3 Block Data Transfers

In the previous section the function `REGISTER_LIST` was related to the ARM6's implementation, which uses a 16-bit mask. This section covers the functions `LDM_LIST` and `STM_LIST`. The following functions are defined in HOL:

```
⊢def REG_WRITE_RP n reg mode mem ireg first =
        REG_WRITE reg mode (RP ldm ireg (MASKN 16 n ireg)) (MEMREAD mem (first + w32 n * w32 4))

⊢def MEM_WRITE_RP n reg mode mem ireg first =
        MEM_WRITE_WORD mem (first + w32 n * w32 4) (REG_READ6 reg mode (RP stm ireg (MASKN 16 n ireg)))
```

The functions `REG_WRITE_RP`/`MEM_WRITE_RP` represent the micro-architecture level load/store operation for the n^{th} word transfered. These are used in the following definitions:

```
⊢def (REG_WRITEN 0 reg mode mem ireg first = reg) ∧
     REG_WRITEN (SUC n) reg mode mem ireg first =
     REG_WRITE_RP n (REG_WRITEN n reg mode mem ireg first) mode mem ireg first

⊢def (MEM_WRITEN 0 reg mode mem ireg first = mem) ∧
     MEM_WRITEN (SUC n) reg mode mem ireg first =
     MEM_WRITE_RP n reg mode (MEM_WRITEN n reg mode mem ireg first) ireg first
```

The functions `REG_WRITEN`/`MEM_WRITEN` give the n^{th} state of the register-bank/memory while performing a block load/store; they are used in constructing and validating the g_c functions in Section 5.1. The final state of the register-bank or memory – as given by the functions h_c – is obtained when the first argument is ℓ. The following lemmas relate these definitions with `LDM_LIST` and `STM_LIST`:

```
⊢ ∀P U base mem reg mode.
        LDM_LIST mem reg mode (FST (ADDR_MODE4 P U base ireg)) =
        REG_WRITEN (LENGTH (REGISTER_LIST ireg)) reg mode mem ireg
          (FIRST_ADDRESS P U base (WB_ADDRESS U base (LENGTH (REGISTER_LIST ireg))))

⊢ ∀P U base mem reg mode.
        STM_LIST mem (SUB8_PC reg) mode (FST (ADDR_MODE4 P U base ireg)) =
        MEM_WRITEN (LENGTH (REGISTER_LIST ireg)) reg mode mem ireg
          (FIRST_ADDRESS P U base (WB_ADDRESS U base (LENGTH (REGISTER_LIST ireg))))
```

The first element of `ADDR_MODE4` is a list of register indices paired with memory addresses (see Section 2.2). The lemmas show that applying the list folding operations `LDM_LIST`/`STM_LIST` to this list is equivalent to applying `REG_WRITEN`/`MEM_WRITEN` with appropriate arguments.

The second lemma accounts for the possibility of storing the program counter. The function `STM_LIST` uses `REG_READ` to access the registers, whereas `MEM_WRITEN` uses `REG_READ6`; the former adds eight to the program counter value, but this is countered by the data abstraction which applies `SUB8_PC`. With load instructions, a series of additional lemmas are required to manipulate (normalise) various combinations of register updates (generated by the *pc*-increment,

write-back, block load and data abstraction operations) and these must take account of whether or not the fifteenth bit of the instruction register is set.

The first address is expressed using the function FIRST_ADDRESS; the ARM6 uses the ALU and an offset to compute this value. The following lemma shows that this computation is correct:

```
⊢ ∀ireg ic base c borrow2 mul.
        1 ≤ LENGTH (REGISTER_LIST (w2n ireg)) ∧
        ((ic = ldm) ∨ (ic = stm)) ⇒
        (FIRST_ADDRESS (WORD_BIT 24 ireg) (WORD_BIT 23 ireg) base
           (WB_ADDRESS (WORD_BIT 23 ireg) base (LENGTH (REGISTER_LIST (w2n ireg)))) =
         SND (ALU6 ic t3 ireg borrow2 mul (OFFSET ic t3 ireg (WORD_BITS 15 0 ireg)) base c))
```

There is a similar lemma to show that the computation of the write-back address, at cycle t_4, is also correct.

5.4 Summary

The formal verification makes use of one-step theorems [5]. The two main verification conditions (theorems two and four on page 9) are tackled using case splitting and the simplifier (term-rewriting). The first level of case splitting is on the instruction class; this means that pre-existing parts of the proof script [6] are used without major alteration.[3] The ARM6 implementation of the block data transfers is symbolically executed using functions h_c; temporal induction is used to prove that these functions evaluate the t_n-phase of execution (Section 5.1). Seven sub-cases are listed in Section 5.1, however, further case splitting is used to reason about particular instruction variants (e.g. with write-back, user mode access or when restoring the CPSR; and also whether the first register of a block store is the base register). The resultant processor states are expressed using functions REG_WRITEN and MEM_WRITEN; these are shown to be related to the function LDM_LIST and STM_LIST using lemmas about priority register masking (Sections 5.2 and 5.3). These and other lemmas are used to relate processor states with those given by the ISA specification.

By including the pipeline state at the ISA level, there was no need to explicitly consider the special cases of writing to the memory addresses $pc + 4$ and $pc + 8$. Using the no-clobber or data forwarding methods [6] would have added to the verification effort.

With the size and complexity of the ARM6 model, it is quite easy for the proof run-times and terms (representing the state of the processor) to become very large. Generating lots of sub-goals, possibly containing large terms, inevitably burdens the individual carrying out an interactive proof. This is mitigated by structuring the proof with the use of lemmas and by being careful in choosing when and how to case split. The method of state evaluation is also of significance; one must decide when to – or, more importantly, not to – expand with a given function definition. Call-by-value conversion is used when evaluating the next state function but this is then combined with the simplifier, which provides contextual re-writing. Although the complexity of the design must be managed it is not an overwhelming problem. The script files for the block data transfer lemmas and the main proof (covering all of the instruction classes) are both approximately a thousand lines long. The overall proof run-time is in the order of a few minutes.

6 Conclusion

This paper has described the work that was involved in extending a partial ARM6 verification [6] to include the block data transfer instruction class. The HOL proof system has been used to construct a concise programmer's model formalisation for this class; this is based on using standard list operations, which are provided in the standard HOL distribution. Daniel Schostak's specification was used as the basis for the HOL model of the ARM6 micro-architecture. The implementation's execute stage is multi-cycled, with a block load taking up to twenty cycles to complete. The instruction timing is determined by the number of registers to be transferred and this is specified by a duration map. In previous verification work with

[3] Some changes were made with the inclusion of the pipeline state at the ISA level. In particular, the single data store proof became simpler.

micro-programmed and pipelined designs [5, 6] the processor control logic has been sufficiently simple that the duration for each instruction is a known constant value. Therefore, additional verification techniques have had to be employed in order to reason about the correctness of the block data transfers. In particular, it was necessary to use induction over time to establish the behaviour of the processor during a sub-stage of the execution. This sub-stage is preceded by two cycles (forming an initial state precondition) and the instruction may complete up to three cycles afterwards. In order to relate the list model with the masking method, a number of auxiliary functions were defined; these enabled inductions to be carried out on the register list length. Functions were defined so as to directly specify the state of the processor part way through the iterated t_n-phase of execution. This paper has presented a number of key lemmas that were used in relating the two different models. The `LEAST` operator was used in specifying the next register to be transfered by the processor; HOL provides a few handy theorems for reasoning about this operator.

This work has demonstrated that the verification strategy (based on symbolic execution) is well suited to adding further instructions to a verified processor design. It was a relatively straightforward task to modify the processor and instruction set models, and much of the pre-existing proof scripts needed little or no modification. This point has been further demonstrated with the verification of the multiply instruction class. The proof run-time for the verification of the block data transfers is longer than for most other instruction classes, but the overall run-time has only increased proportionately i.e. the proof run-time is essentially linear with respect to the number of instruction classes.

To completely verify a commercial processor design one will inevitably have to tackle complex instruction classes such as the block data transfers. This may entail verifying that invariants hold during given phases of instruction execution. This has been shown to be feasible with the HOL model of the ARM6. All core instruction classes have now been verified.

References

[1] Sven Beyer, Chris Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In Daniel Geist and Tronci Enrico, editors, *Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 51–65. Springer-Verlag, 2003.

[2] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Mandayam K. Srivas and Albert Camilleri, editors, *FMCAD '96*, volume 1166 of *LNCS*, pages 275–293. Springer-Verlag, 1996.

[3] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Proceedings of the 6th International Conference, CAV '94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80, Berlin, 1994. Springer-Verlag.

[4] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, June 1989.

[5] Anthony Fox. An algebraic framework for modelling and verifying microprocessors using HOL. Technical Report 512, University of Cambridge, Computer Laboratory, April 2001.

[6] Anthony Fox. Formal verification of the ARM6 micro-architecture. Technical Report 548, University of Cambridge Computer Laboratory, November 2002.

[7] Steve Furber. *ARM: system-on-chip architecture*. Addison-Wesley, second edition, 2000.

[8] Brian T. Graham. *The SECD Microprocessor, A Verification Case Study*. Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1992.

[9] Neal Harman and John Tucker. Algebraic models and the correctness of microprocessors. In George Milne and Laurence Pierre, editors, *Correct Hardware Design and Verification*

Methods, volume 683 of *Lecture Notes in Computer Science*, pages 92–108. Springer-Verlag, 1993.

[10] Neal A. Harman. Verifying a simple pipelined microprocessor using Maude. In M Cerioli and G Reggio, editors, *Recent Trends in Algebraic Development Techniques: 15th Int. Workshop, WADT 2001*, volume 2267 of *Lecture Notes in Computer Science*, pages 128–151. Springer-Verlag, 2001.

[11] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor*, volume 795 of *LNCS*. Springer-Verlag, 1994.

[12] Robert B. Jones, Jens U. Skakkebæk, and David L. Dill. Formal verification of out-of-order execution with incremental flushing. *Formal Methods in System Design*, 20(2):139–158, March 2002.

[13] Jeffrey J. Joyce. Formal verification and implementation of a microprocessor. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 129–157. Kluwer Academic Publishers, 1988.

[14] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.

[15] K. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In A. J. Hu and M. Y. Vardi, editors, *CAV '98*, volume 1427 of *LNCS*. Springer-Verlag, 1998.

[16] Steven P. Miller and Mandayam K. Srivas. Applying formal verification to the AAMP5 microprocessor: A case study in the industrial use of formal methods. *Formal Methods in Systems Design*, 8(2):153–188, March 1996.

[17] Jun Sawada and Warren A. Hunt, Jr. Verification of FM9801: An out-of-order model with speculative execution, exceptions, and program-modifying capability. *Formal Methods in System Design*, 20(2):187–222, March 2002.

[18] David Seal, editor. *ARM Architectural Reference Manual*. Addison-Wesley, second edition, 2000.

[19] Phillip. J. Windley and Michael L. Coe. A correctness model for pipelined microprocessors. In Ramayya Kumar and Thomas Kropf, editors, *TPCD '94*, volume 901 of *LNCS*, pages 33–51. Springer-Verlag, 1995.

Appendix

A Primitive Operations

```
⊢def BITS h l n = n MOD 2^SUC h DIV 2^l
⊢def BIT b n = (BITS b b n = 1)
⊢def WORD_BITS h l n = BITS h l (w2n n)
⊢def WORD_BIT b n = BIT b (w2n n)
⊢def BITV n b = BITS b b n
⊢def SBIT b n = (if b then 2^n else 0)
⊢def (BITWISE 0 op x y = 0) ∧
     BITWISE (SUC n) op x y = BITWISE n op x y + SBIT (op (BIT n x) (BIT n y)) n

⊢def ONECOMP wl n = 2^wl - 1 - n MOD 2^wl
⊢def CLEARBIT wl b a = BITWISE wl (∧) a (ONECOMP wl 2^b)
⊢def (SUM 0 f = 0) ∧ SUM (SUC m) f = SUM m f + f m

⊢def (∀f e. FOLDL f e [] = e) ∧ ∀f e x l. FOLDL f e (x::l) = FOLDL f (f e x) l
⊢def (∀f. MAP f [] = []) ∧ ∀f x l. MAP f (x::l) = f x::MAP f l
⊢def (∀P. FILTER P [] = []) ∧ ∀P h t. FILTER P (h::t) = (if P h then h::FILTER P t else FILTER P t)
⊢def (∀l. EL 0 l = HD l) ∧ ∀n l. EL (SUC n) l = EL n (TL l)
⊢def (ZIP ([],[]) = []) ∧ ∀x1 l1 x2 l2. ZIP (x1::l1,x2::l2) = (x1,x2)::ZIP (l1,l2)
⊢def (∀x. SNOC x [] = [x]) ∧ ∀x x' l. SNOC x (x'::l) = x'::SNOC x l
⊢def (∀f. GENLIST f 0 = []) ∧ ∀f n. GENLIST f (SUC n) = SNOC (f n) (GENLIST f n)
```

B ISA Specification

```
⊢def REGISTER_LIST n = (MAP SND o FILTER FST) (GENLIST (λb. (BIT b n,b)) 16)

⊢def ADDRESS_LIST start n = GENLIST (λa. start + w32 (4 * a)) n

⊢def WB_ADDRESS U base len = (if U then $+ else $-) base (w32 (4 * len))

⊢def FIRST_ADDRESS P U base wb =
        if U then if P then base + w32 4 else base else if P then wb else wb + w32 4

⊢def ADDR_MODE4 P U base n =
         let reg_list = REGISTER_LIST n in
         let len = LENGTH reg_list in
         let wb = WB_ADDRESS U base len in
         let addr_list = ADDRESS_LIST (FIRST_ADDRESS P U base wb) len
         in (ZIP (reg_list,addr_list),wb)
```

```
⊢def LDM_LIST mem reg mode bl_list =
        FOLDL (λreg' (rp,addr). REG_WRITE reg' mode rp (MEMREAD mem addr)) reg bl_list

⊢def STM_LIST mem reg mode bl_list =
        FOLDL (λmem' (rp,addr). MEM_WRITE_WORD mem' addr (REG_READ reg mode rp)) mem bl_list
```

```
⊢def DECODE_LDM_STM n = (BIT 24 n,BIT 23 n,BIT 22 n,BIT 21 n,BIT 20 n,BITS 19 16 n,BIT 15 n)
```

C Addendum to the ARM6 Specification

```
⊢def NBS ic is ireg m =
        if WORD_BIT 22 ireg ∧
          (((is = tn) ∨ (is = tm)) ∧ (ic = ldm) ∧ ¬WORD_BIT 15 ireg ∨
           ((is = t4) ∨ (is = tn)) ∧ (ic = stm))
        then
          usr
        else
          DECODE_MODE m

⊢def MASK nxtic nxtis mask rp =
         if (nxtic = ldm) ∨ (nxtic = stm) then
            if nxtis = t3 then ONECOMP 16 0 else CLEARBIT 16 rp mask
         else
           ARB

⊢def RP ic list mask =
         if (ic = ldm) ∨ (ic = stm) then
           LEASTBIT (BITWISE 16 (∧) list mask)
         else
           ARB

⊢def PENCZ ic list mask =
         if (ic = ldm) ∨ (ic = stm) then
           BITWISE 16 (∧) list mask = 0
         else
           ARB

⊢def OFFSET ic is ireg list =
        if (is = t3) ∧ ((ic = ldm) ∨ (ic = stm)) then
          if WORD_BIT 23 ireg then
            w32 3
          else if WORD_BIT 24 ireg then
            w32 (4 * (SUM 16 (BITV list) - 1) + 3)
          else
            w32 (4 * (SUM 16 (BITV list) - 1))
        else if (is = t4) ∧ ((ic = ldm) ∨ (ic = stm)) then
          w32 (4 * (SUM 16 (BITV list) - 1) + 3)
        else if (is = t5) ∧ ((ic = br) ∨ (ic = swi_ex)) then
          w32 3
        else
          ARB
```

satGSTE: Combining the Abstraction of GSTE with the Capacity of a SAT Solver

Jin Yang, Eli Singerman, Rami Gil

Design Technologies

Intel corporation

Overview

- G/STE Overview and Motivation
- Spec language - Assertion Graphs (AG)
- satGSTE – Bounded model checking of AG
- satGSTE algorithm
- Results
- Future directions

STE : formal verification through symbolic simulation (Seger & Bryant)

- STE - Symbolic Trajectory Evaluation
  - a lattice-based model checking technology based on a form of symbolic simulation (Seger & Bryant)
  - proven high capacity with a high degree of automation
  - a comprehensive methodology
- Active Industrial Usage in Hardware Verification
  - Intel
    - Many models
    - floating point arithmetic unit against IEEE 754
    - Instruction length decoder unit
  - Motorola
    - memory units with million transistors
  - IBM
  - Compaq

STE: Simple Specification

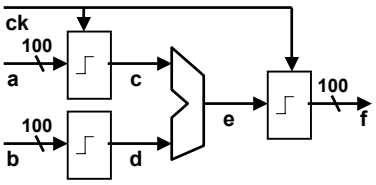
- Normalized basic STE assertion

A_0 & N(A_1) & … & N^{k-1}(A_{k-1}) Antecedent (stimulus)

==> C_0 & N(C_1) &…& N^{k-1}(C_{k-1}) Consequent (response)

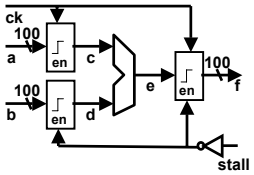
  - e.g.

(!ck & a[99:0]=A[99:0] & b[99:0]=B[99:0]) &
N(ck) & N^2(!ck) & N^3(ck)
==> N^3(c[99:0]=A[99:0]+B[99:0])

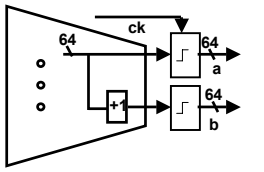


STE: Limitation

- Very limited expressiveness
  - ⊙ what about stall?
    - ○ stall causes arbitrary long latency



  - ⊙ what about backward analysis?
    - ○ simulation is forward in nature



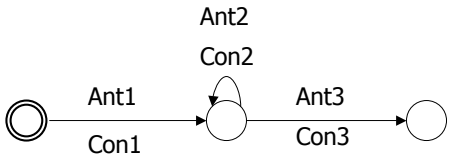
(a=ptr) ==> (b = ptr+1)

GSTE: Generalized STE (Yang & Seger)

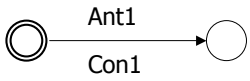
- Extends STE to verify properties over indefinitely long time intervals
- Has very high capacity thanks to built-in abstract representation of states (this comes at a cost, though…)
- GSTE properties are assertion graphs (AG), with each edge in the graph labeled with an STE (antecedent, consequent) pair

AG (Assertion Graphs)

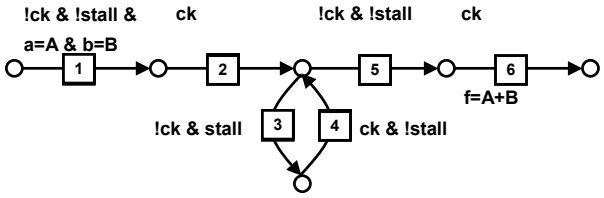
- **AG is a graph consisting of nodes and edges where each edge has**
  - **Antecedent**
  - **Consequent**

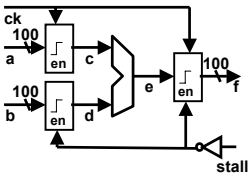


- Loops etc. provide infinite-time properties (fairness can be imposed)
- STE is a special case of GSTE where AG is simply a single edge graph



GSTE: Model Checking





- **Symbolic Simulation Phase:**

edge	1	2	3	4	5	6
simulation result	!ck & !stall & a=A & b=B	ck & c=A & d=B & e=A+B	!ck & stall & c=A & d=B & e=A+B !ck & stall & c=A & d=B & e=A+B	ck & stall & c=A & d=B & e=A+B	!ck & !stall c=A & d=B & e=A+B !ck & !stall c=A & d=B & e=A+B	ck & f=A+B

"Fix-point computation":
repeat updating the simulation result until no further change

satGSTE – Bounded MC of AG

- Motivation
  - Enable fast AG spec debugging/refinement – optimized for failure
  - Enable “light weight” verification sessions -- exercise chosen paths in the AG
  - Efficient full model-checking of a-cyclic graphs
- Method
  - Extract all linear paths up to a given bound
  - Symbolically simulate each path
  - Generate verification conditions
  - Submit to a SAT Solver

Algorithm: high level description

- Extract paths from user AG
  - For every path
  - For every edge
    - Simulate circuit by driving the antecedents
    - Get symbolic values of every consequent node
    - Build a formula stating that the consequent fails
    - Send to SAT solver
    - If SAT says NO continue, else
  - Create a linear counter example from SAT assignment
- All paths up to bound passed - no counter example found for this bound.

satGSTE – path extraction

- Path extraction is the first phase of sarGSTE
- Paths are held in a tree structure
- For performance, it is crucial to optimize
  - Compact tree structure –maximum sharing of paths by prefixes
  - Dynamically drop illegal paths (e.g., in compositional AG – when a condition is not met)
  - Allow an option of maximum edge coverage as an alternative to all paths of up to a given bound

satGSTE – path simulation

- Is done with bSTE – a version of STE that uses bexpr rather than BDD as the underline data structure for boolean expressions
  - Bexprs are not canonical, but have some sharing and allow much bigger capacity (pay the price in SAT time, though)
- Exploits the aggressive STE abstraction

satGSTE: counter example presentation

- satGSTE presents a linear counter example retrieved from the values returned by the SAT solver
  - ⊙ Go back from the failing edge to previous edges on the failing path
  - ⊙ Justify the linear counter example on every edge
  - ⊙ Counter example information is held in an explanation tree data structure
- CEX information can be accessed via interface function by the user or by GUI tools

satGSTE: Results

- Were run on real Intel circuits (50K latches)
- As expected, failures (CEX) are found very fast
- Provided full proof for a-cyclic AGs

Test	Bound	satGSTE time	satGSTE memory	# nodes	Outcome
Ckt1	3	49.8	145 MB	300	Pass
Ckt2	3	1 sec	40 MB	34130	Pass
Ckt3	12	4 sec	40 MB	879	CEX
Ckt4	1	4.2 sec	80 MB	297954	Full
Ckt5	1	24.1 sec	200 MB	46580	CEX
Ckt6	3	24 sec	230 MB	533421	Pass
Ckt7	3	6.2 sec	40 MB	6299	Pass

Future Directions

- Further simplifications of path extraction algorithm in satGSTE -- better sharing
- Improve bexpr sharing and partial evaluation (simplification)
- Add interactive mode for running and CEX debugging.
- A hybrid SAT/BDD algorithm for GSTE (leverage from recent research on fixed-point via SAT)

Symbolic Trajectory Evaluation using Satisfiability Solvers DRAFT

Koen Claessen and Jan-Willem Roorda

Chalmers University of Technology, Sweden

Abstract. In this article we describe a hierarchy of semantics for STE assertions. For several semantics we give a sound and complete algorithm based on a SAT-solver for three-valued logic. Finally, we show how we can adapt a standard DPLL-style SAT-solver to a SAT solver for three-valued logic.

1 Introduction

Symbolic Trajectory Evaluation (STE) [10] is a high-performance simulation-based model checking technique. It combines *three-valued simulation* (using the standard values 0 and 1 together with the extra value X, "don't know") with *symbolic simulation* (using symbolic expressions to drive inputs). STE has been extremely successful in verifying properties of circuits containing large data paths (such as memories, fifos, floating point units, etc.), which are beyond the reach of traditional symbolic model checking techniques [13, 2, 9].

STE assertions are of the form $A \Longrightarrow C$; the antecedent A drives the simulation, and the consequent C expresses the conditions that should result. The antecedent A and the consequent C are formulas in a restricted temporal logic called *Trajectory Evaluation Logic* (TEL), a subset of Linear Temporal Logic (LTL). The only variables in TEL (called *symbolic variables*) are taken from a set of time independent two-valued variables. Circuit nodes can be specified to have the values 0 and 1 or a symbolic value given as a propositional formula in terms of the symbolic variables. Formulas can be combined using conjunction **and** and the the next time operator **N**. There is no disjunction or negation in the logic. A typical STE assertion for a memory circuit is:

$$\begin{array}{l} \quad wr \textbf{ is } 1 \quad \textbf{and } rd \textbf{ is } 0 \textbf{ and } addr \textbf{ is } A \textbf{ and } in \textbf{ is } V \textbf{ and} \\ \textbf{N}(\ wr \textbf{ is } 0 \quad \textbf{and } rd \textbf{ is } 1 \textbf{ and } addr \textbf{ is } A\ ) \\ \quad \Longrightarrow \\ \textbf{N}(\ out \textbf{ is } V\ ) \end{array}$$

The assertion states that if we write a value V to an address A, reading the address A at the next moment in time should yield the same value V. The standard implementation of STE uses a simulator with takes BDD-based symbolic

expressions resulting in 0, 1 or X as signal type. Circuit nodes that are not specified by the antecedent at a certain time point are assumed to have the unknown value X at that time. (For instance, in the case of the memory assertion, the initial values of the memory locations are all assumed to be X). This assumption makes STE very efficient, because is very cheap to simulate parts of the circuit that are fed by unknown inputs, as the BDD can simply return the value X for the nodes of that part. In fact, the complexity of checking STE assertions largely depends on the number of symbolic variables used in the assertion rather then on the complexity of the circuit.

The disadvantage of this use of three-valued simulation is that it can yield the value X for an circuit node present in the result, while two-valued simulation would have yielded a 0 or a 1. This means that some STE assertions are not provable in STE, while they are true in the LTL semantics. Thus, the STE gives a less precise semantics to a subset of LTL, which means that some STE assertions are true in a standard LTL-semantics, while they are not true in the STE semantics. The STE semantics is less precise than the LTL semantics in two respects:

- The semantics of STE uses three-valued reasoning; all variables that are not explicitly given a binary value, are allowed to take on the value X. This leads for example to the STE assertion True $\Longrightarrow$ *out* **is** 1 not being true for the circuit specified by $out = in \vee \neg in$. Of course, the assertion is true in the standard semantics for LTL.
- The semantics of STE only propagates information in a forwards fashion. For example, the STE assertion $\mathbf{N}(out\,\mathbf{is}\,1) \Longrightarrow in\,\mathbf{is}\,1$, for a circuit that consists of a single delay element with input *in* and output *out* is not true is STE, while it is clearly true in the LTL semantics.

A property that holds in LTL but does not hold in STE can always be transformed into an LTL-equivalent property that is true in STE by introducing more symbolic variables. For instance in the first example above, we can add a symbolic variable i to drive the input *in* in the assertion, so that we arrive at the assertion $in\,\mathbf{is}\,i \Longrightarrow out\,\mathbf{is}\,1$ which is true in STE. The user of an STE system can in this way balance the line between weak reasoning power but quick verification results, and strong reasoning power but more expensive checks.

In this article, we describe a hierarchy of semantics for STE properties. At the top of the hierarchy is the standard LTL-like semantics, where models directly correspond to runs of the circuit. At the bottom of the hierarchy is the standard STE semantics (as defined elsewhere [10]), where models are abstract representations of what might be runs of the circuit. The hierarchy makes the balance between proof power and efficiency explicit.

Furthermore, we give a SAT based algorithm for checking STE properties. The algorithm is able to prove backwards properties which are not provable by simulator based algorithms, and is based on a SAT solver for three valued logic. We show how we can adapt a standard DPLL style SAT solver [11] to a SAT solver for three valued logic.

In the rest of the paper, we start by defining the hierarchy of semantics for STE assertions, and proceed by giving implementations corresponnding to semantics in this hierarchy. We finish with some experimental results and conclusions.

2 A Hierarchy of Semantics for STE Properties

In this section, we discuss a hierarchy of different semantics for STE properties. At the top of the hierarchy is the standard LTL-like semantics, where models directly correspond to runs of the circuit. At the bottom of the hierarchy is the standard STE semantics (as defined elsewhere [10]), where models are abstract representations of what might be runs of the circuit.

2.1 Circuits

A *circuit* c consists of a set of nodes $\mathcal{N}$ connected by logical gates and registers. We distinguish the following subsets of the set of nodes: $\mathcal{I}$ is the set of input nodes, $\mathcal{S}$ is set of state holding nodes, used to model registers, and $\mathcal{O}$ is the set of observable nodes of the circuit. The node sets $\mathcal{I}$ and $\mathcal{S}$ are determined by the circuit, the set of observable nodes $\mathcal{O}$ can be chosen to be any arbitrary subset of all circuit nodes. We also assume that for every node n in $\mathcal{S}$, there is a node n' in $\mathcal{N}$ which models the value of that node in the next state. The set of all such n' is called $\mathcal{S}'$.

It is common to describe a circuit in the form of a netlist. We will do this here too, but we will not mention registers explicitly in the netlist. Instead, we will explicitly mention n and n' in the netlist. For simplicity, the only gates we allow in netlists are AND-gates and (implicitly) inverters. We know this is enough to represent any other logical function without a blow-up in extra logic. It is however straightforward to extend this notion of netlist to include more operations. This is not needed for the rest of the paper.

To define how we represent netlists exactly, we need the following preliminaries. A *node reference*, written m, is either a node n or an inverted node $\bar{n}$. A *definition* is a triple of node references, written $m_1 = m_2 \text{ AND } m_3$. Here, we call the node of m_1 the *head* of the definition, and the nodes of m_2 and m_3 the *operands*.

Definition 1 (Netlist). *Given a circuit c with $\mathcal{N}$ divided up as above, a* netlist *for c is a finite set of definitions, such that*

- *For all nodes n in $\mathcal{N} - (\mathcal{I} \cup \mathcal{S})$ there is exactly one definition with head n,*
- *The definitions are acyclic, i.e. they can be ordered in a list such that no operands of a definition appear as a head later in the list.*

2.2 Values

The values that circuit nodes can have are represented by the set $\mathbb{V}$. The set $\mathbb{V}$ contains as a subset the set $\mathbb{B}$ of regular values 0 and 1, but also a special value

x	$\neg x$
0	1
1	0
X	X

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1
X	0	0
0	X	0
X	1	X
1	X	X
X	X	X

Fig. 1. Three-valued extensions of the gates

X, which is the *unknown* value. The idea is that we can get an approximation of the behavior of a circuit by using the abstract value X as input. If the result is a 0 or a 1 anyway, we know that this result is independent of whether we we will use a 0 or a 1 as input.

Formalising this, we introduce a partial ordering on $\mathbb{V}$. We let $\mathrm{X} \leq 0$ and $\mathrm{X} \leq 1$, but we leave 0 and 1 incomparable. Now, we have to extend the normal boolean gates in order to deal with the extra value X. They can be extended in many ways, but there is one property that the extensions must fulfill: They have to be *monotonic* with respect to the partial ordering $\leq$ on $\mathbb{V}$. The reason for this is that a gate cannot give a concrete answer (0 of 1) when one of its inputs is X and then change its mind when that input is made more concrete. The way of extending operators such that they lose the least amount of information is given in Figure 1.

We model circuits by their *transition functions*. Given a circuit c, a transition function F_c takes as input values for the input and state nodes, and yields values for the observable nodes and the state nodes at the next moment in time. Thus we define $I = (\mathcal{I} \to \mathbb{V})$, $S = (\mathcal{S} \to \mathbb{V})$, and $O = (\mathcal{O} \to \mathbb{V})$. The type of a transition function then becomes $F_c : I \times S \to O \times S$.

Given the netlist of a circuit, we can easily construct a transition function that models the circuit. For example, we can interpret each definition in the netlist as a three-valued gate as described in Figure 1. Given the values for all nodes in $\mathcal{I}$ and $\mathcal{S}$, we can simply evaluate the values for all nodes. We denote this transition function by F_c^*. However, there are also other ways which are correct, as long as the resulting transition function is monotonic. For the rest of this paper we assume that F_c is a monotonic function that is at least as precise as F_c^*.

2.3 Properties

Trajectory Evaluation Logic (TEL) is a restricted linear temporal logic in which properties over bounded time intervals can be expressed. The only variables in the logic are time-independent *propositional variables* taken from a set V.

Definition 2 (Trajectory Evaluation Logic). *Given a set of symbolic variables V, the language of* Trajectory Evaluation Logic *is given by:*

$$\begin{array}{rl} f ::= & n \textbf{ is } 0 \\ | & n \textbf{ is } 1 \\ | & f_1 \textbf{ and } f_2 \\ | & P \rightarrow f \\ | & \mathbf{N}f \end{array}$$

where $n \in \mathcal{N}$ and P is a propositional formula over the set of propositional variables V.

A key property of Trajectory Evaluation Logic is that a (consistent) formula specifies for every time instance t and every node n, the set of values it is allowed to take. This property would no longer hold if we were to add negation or disjunction to the logic. This seems limiting, but it is exactly this property which allows implementations of STE to model the value of a node by a BDD in terms of the propositional variables.

We define the meaning of a TEL formula by a satisfaction relation. Models in this relation are sequences σ of assignments of values to nodes over time. More formally, $\sigma : \mathbb{N} \rightarrow \mathcal{N} \rightarrow \mathbb{V}$. We define a time shifting operator σ^1 by $\sigma^1(t)(n) = \sigma(t+1)(n)$. We denote standard propositional satisfiability by $\models_{\text{Prop}}$.

Definition 3 (Satisfaction of trajectory evaluation logic formulas). *Satisfaction of a trajectory evaluation logic formula f, by a sequence σ, and a valuation $\phi : V \rightarrow \mathbb{B}$ (written $\phi, \sigma \models f$) is defined by*

$$\begin{array}{lcl} \phi, \sigma \models n \textbf{ is } 0 & \equiv & \sigma(0)(n) = 0 \\ \phi, \sigma \models n \textbf{ is } 1 & \equiv & \sigma(0)(n) = 1 \\ \phi, \sigma \models f_1 \textbf{ and } f_2 & \equiv & \phi, \sigma \models f_1 \textit{ and } \phi, \sigma \models f_2 \\ \phi, \sigma \models P \rightarrow f & \equiv & \phi \models_{\text{Prop}} P \textit{ implies } \phi, \sigma \models f \\ \phi, \sigma \models \mathbf{N}f & \equiv & \phi, \sigma^1 \models f \end{array}$$

A *trajectory assertion*, notation $A \Longrightarrow C$, is a pair of TEL formulas A and C, where A represents a set of assumptions made, and C represents set of consequences that is believed to follow from the assumptions.

In the following, we will give a hierarchy of semantics that relate circuits to trajectory assertions. Some of these semantics will be *approximations*, which means that a trajectory assertion might be classified as false, though in reality one would consider it to be true. However, if an assertion is classified to be true, it will also be true in reality. The hope is that an approximative semantics will be easier to check than a more precise semantics.

2.4 Behaviour Semantics

The first semantics we give, the *behaviour semantics*, is not an approximation, but characterises exactly when a trajectory assertion is true or false for a given

circuit. The behaviour semantics therefore is the semantics which all other semantics have to comply with.

First, we define the concept of behaviors, which are all sequences which correspond to real runs of the circuits, i.e. sequences that respect the transition function, and only map nodes to the values 0 or 1.

Definition 4 (Behaviour). *Given a circuit c with transition function F_c. A sequence σ is a* behaviour *iff for all times $t \in \mathbb{N}$ and for all nodes $n \in \mathcal{N}$ it holds that*

$$\sigma(t)(n) \in \mathbb{B},$$

and for all $t \in \mathbb{N}$ it holds that

$$(\sigma_{\mathcal{S}}(t+1), \sigma_{\mathcal{O}}(t)) = F_c(\sigma_{\mathcal{I}}(t), \sigma_{\mathcal{S}}(t)).$$

Here, we use the notation σ_N to denote the function σ with the domain restricted to the nodes N.

The behaviour semantics corresponds to an LTL-like semantics for circuits, i.e. a circuit satisfies an assertion if and only if all circuit behaviours satisfy the assertion.

Semantics 1 (Behaviour Satisfiability) *A circuit c* behaviour satisfies *a trajectory assertion $A \Longrightarrow C$, written $c \models_{\text{Beh}} A \Longrightarrow C$ iff for every valuation $\phi \in V \to \mathbb{B}$ and for every behaviour σ of c, it holds that*

$$\phi, \sigma \models A \;\;\Rightarrow\;\; \phi, \sigma \models C.$$

In order to be able to compare different semantics with each other, we introduce the following semantics precision ordering.

Definition 5 (Semantics Precision Ordering). *Given two semantics relations $\models_1$ and $\models_2$, relating circuits to trajectory assertions, we say that $\models_1$ is* less precise *than $\models_2$, notation $\models_1 \preceq \models_2$, iff, for all circuits c and trajectory assertions $A \Longrightarrow C$ it holds that*

$$c \models_1 A \Longrightarrow C \;\;\Rightarrow\;\; c \models_2 A \Longrightarrow C.$$

It is easy to see that the precision ordering on semantics is reflexive and transitive. All other semantics for trajectory assertions are supposed to be at most as precise as the behaviour semantics.

2.5 Three-Valued Semantics

The *three-valued semantics* is the first approximative semantics we introduce in this paper. Here, we will make use of the third value X, which stands for the unknown value.

First, we define the set of X-behaviours, which is a superset of the set of behaviours of a circuit. In X-behaviours nodes are allowed to take the value X, whereas in behaviours nodes can only take the values 0 and 1.

Definition 6 (X-Behaviour). *Given a circuit c with transition function F_c. A sequence σ is an* X-behaviour *iff for all $t \in \mathbb{N}$ it holds that*

$$(\sigma_{\mathcal{S}}(t+1), \sigma_{\mathcal{O}}(t)) = F_c(\sigma_{\mathcal{I}}(t), \sigma_{\mathcal{S}}(t)).$$

In the three-valued satisfiability relation, we require that all X-behaviours of the circuit satisfy the assertion, in order for an assertion to be true. This leads to a semantics in which fewer assertions are valid. However, if the assertion is valid in the three-valued semantics, it is also valid in the behaviour semantics.

Semantics 2 (Three-Valued Satisfiability) *A circuit c* three-valuedly satisfies *a trajectory assertion $A \Longrightarrow C$, notation $c \models_{\text{Three}} A \Longrightarrow C$ iff for every valuation $\phi \in V \to \mathbb{B}$ and for every* X*-behaviour σ of c, it holds that*

$$\phi, \sigma \models A \quad \Rightarrow \quad \phi, \sigma \models C.$$

The reason why the three-valued semantics is interesting at all, is that it might be cheaper to check three-valued validity of assertions than behaviour validity. The price we pay for this possible loss in complexity is that the semantics becomes less precise.

Proposition 1. *Three-valued satisfiability is less precise than behaviour satisfiability, i.e. $\models_{\text{Three}} \preceq \models_{\text{Beh}}$.*

Moreover, three-valued satisfiability is strictly less precise than behaviour satisfiability. The reason for this is that if a node n is unconstrained by a trajectory formula A at a particular point in time, then in order for the assertion $A \Longrightarrow C$ to hold in the three-valued semantics, every trajectory that gives the value X to node n must satisfy C if it satisfies A. This is a very strong condition – allowing n to be X might lead to too many X's in other parts of the circuit, even though setting n to 0 or 1 would lead to situations that fulfill C.

However, even though there are assertions $A \Longrightarrow C$ which are valid in the behaviour semantics but not in the three-valued semantics, we can always adapt such an assertion to an assertion $A' \Longrightarrow C'$ which is satisfiability equivalent to $A \Longrightarrow C$ in the behaviour semantics, but nevertheless holds in the three-valued semantics. We simply do this by introducing more propositional variables which restrict certain inputs.

2.6 Forwards Semantics

Looking at the standard semantics given to STE assertions, there is yet another source of information loss. In the three-valued semantics above we make use of X-behaviours, which are sequences that are supposed to be fully consistent with the transition function F_c. However, in the standard STE semantics, information known about a node n in a particular point in time, only has to be consistent

with the events that come after than point in time. So, we define yet another semantics, which captures the notion of this *forwards propagation.*

The notion of a sequence that only has to respect forwards propagation is called *trajectory* in the STE literature.

Definition 7 (Trajectory). *Given a circuit c with transition function F_c. A sequence σ is a* trajectory *iff for all $t \in \mathbb{N}$ it holds that*

$$(\sigma_{\mathcal{S}}(t+1), \sigma_{\mathcal{O}}(t)) \geq F_c(\sigma_{\mathcal{I}}(t), \sigma_{\mathcal{S}}(t)).$$

Here, we make use of the ordering $\geq$, which is the ordering from the value set $\mathbb{V}$ lifted to tuples and functions. In other words, when picking values for nodes in time point $t+1$, we have to at least respect what was predicted about them in time point t. For example, if given the values of state and input nodes at time n, the transition function F_c, predicts the value X for a certain node, the trajectory is allowed to take the value $0, 1$ or X for that node. If, however, the F_c the transition function predicts the value 0 or 1, the trajectory is required to take that value.

We define the notion of forwards satisfiability accordingly.

Semantics 3 (Forwards Satisfiability) *A circuit c* forwards satisfies *a trajectory assertion $A \Longrightarrow C$, written $c \models_{\text{Forw}} A \Longrightarrow C$ iff for every valuation $\phi \in V \to \mathbb{B}$ and for every trajectory σ of c, it holds that*

$$\phi, \sigma \models A \quad \Rightarrow \quad \phi, \sigma \models C.$$

The forwards semantics is less precise than the three-valued semantics, and therefore also less precise than the behaviour semantics.

Proposition 2. *Forwards satisfiability is less precise than three-valued satisfiability, i.e. $\models_{\text{Forw}} \preceq \models_{\text{Three}}$.*

The information loss can be easily seen in an assertion $A \Longrightarrow C$ where A says something about an output at a later time than when C says something about an input. For instance, the assertion $\mathbf{N}(\mathit{out}\ \mathbf{is}\ 1) \Longrightarrow \mathit{in}\ \mathbf{is}\ 1$, for a circuit that consists of a single delay element with input *in* and output *out* is not true in the forward semantics.

Again, an assertion $A \Longrightarrow C$ that is valid in the behaviour semantics but not in the forwards semantics, can be adapted to an assertion $A' \Longrightarrow C'$ which is satisfiability equivalent to $A \Longrightarrow C$ in the behaviour semantics, but holds in the forwards semantics. For instance, the assertion $\mathbf{N}(\mathit{out}\ \mathbf{is}\ 1) \Longrightarrow \mathit{in}\ \mathbf{is}\ 1$ can be transformed to $N(\mathit{out}\ \mathbf{is}\ 1)\ \mathbf{and}\ \mathit{in}\ \mathbf{is}\ p \Longrightarrow \mathit{in}\ \mathbf{is}\ 1$ by adding an extra symbolic variable. The assertion $N(\mathit{out}\ \mathbf{is}\ 1)\ \mathbf{and}\ \mathit{in}\ \mathbf{is}\ p \Longrightarrow \mathit{in}\ \mathbf{is}\ 1$ is true in the forwards semantics as the consequent is valid is we set $p = 1$ and the antecedent is valid if we set $p = 0$.

2.7 Y-Semantics

The traditional STE semantics makes use of a *next state function*, denoted by $Y : N \to N$. Thus, it does not distinguish between inputs, outputs, and states. The problem with using this Y function to define the semantics of STE is that it does not really correspond to what actual implementations do. For example, a Y function can only propagate information to the next point in time, whereas actual implementations of STE can propagate information from inputs of circuits directly to outputs of circuits in the same point in time.

The purpose of the next state function is to take information about the current state and compute information about the next state. Naturally, the input nodes in the next state will be mapped to X.

For completeness, we define here the relationship between the traditional STE semantics and the semantics we have discussed so far. We begin with relating transition functions F_c to next state functions Y_c.

Definition 8 (Y-Consistency). *A next state function $Y_c : N \to N$ is* consistent *with a transition function $F_c : I \times S \to O \times S$ iff for all n, n' in N, i in I, o, o' in O, s, s', s'' in S, if it holds that*

$$(o, s') = F_c(i, s) \quad \text{and} \quad (o', s'') = F_c(I_{\mathrm{X}}, s'),$$

then it also holds that

$$Y_c(i \cup o \cup s) = (I_{\mathrm{X}} \cup o' \cup s').$$

Here, we write S_{X} for the constant function that maps all elements in S to X. As we can see, it is a bit awkward to relate F_c to Y_c, since Y_c computes values of nodes belonging to two points in time: first Y_c needs to compute the values of the next state s' and then it needs to compute the values of the other nodes o' belonging to the next state.

Given a Y function, we can define the notion of Y-trajectory.

Definition 9 (Y-trajectory). *A sequence $\sigma : \mathbb{N} \to \mathcal{N} \to \mathbb{V}$ is a Y-trajectory iff for all $t \in N$, it holds that*

$$\sigma(t + 1) \geq Y(\sigma(t)).$$

The definition of Y-satisfiability follows from the definition of Y-trajectories in a natural way.

Semantics 4 (Y-Satisfiability) *A circuit c* Y-satisfies *a trajectory assertion $A \Longrightarrow C$, written $c \models_{\mathrm{Y}} A \Longrightarrow C$ iff for every valuation $\phi \in V \to \mathbb{B}$ and for every Y-trajectory σ of c, it holds that*

$$\phi, \sigma \models A \;\; \Rightarrow \;\; \phi, \sigma \models C.$$

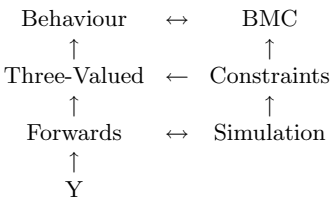


Fig. 2. Semantics and implementation hierarchy

Note that because Y-trajectories do not place constraints on the relations between nodes at the same time step it is not possible to validate assertions that require a flow of information from input to output in the same time step. For instance, the assertion in **is** $0 \Longrightarrow out$ **is** 1 for an inverter gate is not true in the Y-semantics.

Proposition 3. *Y-satisfiability is less precise than forwards satisfiability, i.e.* $\models_{\mathrm{Y}} \preceq \models_{\mathrm{Forw}}$.

2.8 Summing Up

In this section, we have constructed a hierarchy of semantics for trajectory assertions. The most precise semantics, the behaviour semantics, is at the top of the hierarchy, and the least precise semantics, the Y semantics, at the bottom.

The idea is that less precise semantics might have easier decision procedures associated with them. In the next section, we will discuss a number of such decision procedures.

In Figure 2, the hierarchy of the different semantics plus associated decision procedures is shown. An arrow $\uparrow$ with a semantics A below and semantics B above means that A is (strictly) less precise than B. A double arrow $A \leftrightarrow B$ between a semantics and a decision procedure means that the procedure is sound a complete with respect to the semantics. An arrow $\leftarrow$ to the left denotes soundness. Finally, an arrow $\uparrow$ with a procedure A below and a procedure B above means that A can prove (strictly) fewer assertions than B.

3 Decision Procedures

In this section we introduce decision procedures for STE assertions based on three valued SAT solvers. We start with our own decision procedure which is sound w.r.t. the three-valued semantics, but more precise than the forwards semantics.

3.1 Three-valued constraints

In this subsection we give a decision procedure that, from a circuit c and an STE assertion $A \Longrightarrow C$, generates a set of constraints over three-valued variables. These constraints can then be solved by a three-valued constraint solver, a possible implementation of which is described in the next section.

Definition 10 (Three-valued Constraint Problem). *A* three-valued constraint problem *consists of a set of two-valued variables V_2, which are called the* symbolic variables*, a set of three-valued variables V_3, which are called the* node variables*, and a set of* clauses*. A* clause *is a disjunction of one, two or three* literals*. A* literal *is a variable x or a negated variable $\bar{x}$.*

So, a three-valued constraint problem contains a mixture of two-valued and three-valued variables. We need to define what the models of a three-valued clause are. In the following, we will use three-valued assignments, which are functions $\phi : V_2 \cup V_3 \rightarrow \mathbb{V}$. For a negated variable, we define $\phi(\bar{x}) = 0$ if $\phi(x) = 1$ and $\phi(\bar{x}) = 1$ if $\phi(x) = 0$.

Definition 11 (Satisfaction of Three-Valued Clauses). *Let p, q, and r be literals.*

1. *A three-valued clause p is satisfied by a value assignment ϕ iff $\phi(p) = 1$.*
2. *A three-valued clause $p \vee q$ is satisfied by a value assignment ϕ iff $\phi(p) = 0$ implies $\phi(q) = 1$, and $\phi(q) = 0$ implies $\phi(p) = 1$.*
3. *A three-valued clause $p \vee q \vee r$ is satisfied by a value assignment ϕ iff $\phi(p) = 0$ and $\phi(q) = 0$ imply $\phi(r) = 1$, and $\phi(p) = 0$ and $\phi(r) = 0$ imply $\phi(q) = 1$, and $\phi(q) = 0$ and $\phi(r) = 0$ imply $\phi(p) = 1$.*

So, for example, a clause $p \vee q \vee r$ is satisfied when all p, q, r are X, or when two of p, q, r are X, but not when p is X and q and r are 0.

3.2 Constraint variables

Given a circuit c and an STE assertion $A \Longrightarrow C$, we construct three sets of constraints: (1) the set of constraints for the requirements placed by the antecedent, written $[A]$, (2) the set of constraints for the requirements placed by the consequent, written $[C]$, and (3) the set of constraints for the behaviour of the circuit, written $[c]$. We then check, using a three-valued SAT-solver whether every model of the constraint sets $[A]$ and $[c]$ is also a model of $[C]$. If this is the case, then we can conclude that the assertion $A \Longrightarrow C$ holds.

The first step in designing a SAT-based decision procedure for STE assertions is to realize that only the first d steps of a sequence determine whether it satisfies a formula of *depth d*. The depth of a formula is the number of nested uses of **N**.

Proposition 4. *Let d be the depth of a TEL formula f in trajectory evaluation logic, $\phi : V \to \mathbb{V}$ a valuation function and $\sigma, \sigma' : \mathbb{N} \to \mathcal{N} \to \mathbb{V}$ be sequences then:*

$$\phi, \sigma \models f \;\; \textit{and} \;\; \sigma(\{0, \ldots, d\}) = \sigma'(\{0, \ldots, d\}) \;\; \Rightarrow \;\; \phi, \sigma' \models f.$$

Because only the first d time steps of a sequence determine whether an STE assertion is valid, we only consider these time steps in the generation of three-valued constraints. We introduce a set of node variables n_t with $n \in \mathcal{N}$ and $0 \leq t \leq d$ which give the values of the nodes in the circuit over over the first d time steps, we denote this set by $\mathrm{Var}_{\mathcal{N}}$. Furthermore, we introduce symbolic variables for every propositional variable in the assertion, notation $\mathrm{Var}_{\mathrm{V}}$. We define $\mathrm{Var} = \mathrm{Var}_{\mathrm{V}} \cup \mathrm{Var}_{\mathcal{N}}$. In the three-valued semantics nodes can have the value X next to the values 0 and 1, therefore in the variables in the set $\mathrm{Var}_{\mathcal{N}}$ can take values $0, 1$ and X, whereas the variables in the set of symbolic variables $\mathrm{Var}_{\mathrm{V}}$ can only take the values 0 and 1.

3.3 Generating constraints

From a circuit c we can derive a set of three-valued constraints, called the *transition constraints*, corresponding to the behavior of the circuit.

Definition 12 (Transition Constraints). *For each definition $m_1 = m_2$ AND m_3 in the netlist of a circuit c, if m_1 is a reference to a non inverted node n, then we define l_1 to be positive literal of the node n, otherwise l_1 is negative literal. The literals l_2 and l_3 are defined likewise. We introduce the following constraints for the definition.*

$$\begin{array}{c} l_1 \vee \bar{l_2} \vee \bar{l_3} \\ \bar{l_1} \vee l_2 \\ \bar{l_1} \vee l_3 \end{array}$$

We denote the set of all such constraints of definitions in c by $[[c]]$.

Note that for this definition of satisfaction, the constraint for a definition $m = k$ AND l is satisfied by all the assignments of m, k and l in the truth table of the AND gate.

Further, we generate constraints for the assertions A and C as follows.

Definition 13 (Defining Constraints). *We define a mapping $[[\_]](\_)$ from formulas and time points to three-valued constraints as follows:*

$$\begin{array}{ll} [[n \textbf{ is } 0]](t) & \equiv \{\bar{n_t}\} \\ [[n \textbf{ is } 1]](t) & \equiv \{n_t\} \\ [[f_1 \text{ and } f_2]](t) & \equiv [[f_1]](t) \cup [[f_2]](t) \\ [[P \to f]](t) & \equiv P \rightsquigarrow [[f]](t) \\ [[\mathbf{N}f]](t) & \equiv [[f]](t+1) \end{array}$$

The defining constraints *of f, notation $[f]$, are denoted by $[[f]](0)$.*

Here, we use the operator $\rightsquigarrow$, which has the following meaning. Given a clause set C and a propositional formula P, we pick a new two-valued variable p, and clausify the propositional formula $P \rightarrow p$ in the usual way, leading to a set of clauses D. The result of $P \rightsquigarrow C$ then becomes the set of constraints $D \cup \{\bar{p} \vee c | c \in C\}$. In other words, in situations where P is true, the constraints in C have to hold (since p must be 0 then), and in situations where P is false, they do not necessarily have to hold.

3.4 Properties of the constraints

We define the valuation σ_d corresponding to sequence σ by $\sigma_d(n_t) = \sigma(n)(t)$, for $0 \leq t \leq d$. As σ_d contains exactly the same information as σ for the first d time steps, σ is a model of TEL-formula f of depth less or equal to d iff σ_d satisfies its corresponding constraint $[f]$. This is stated in the lemma below.

Lemma 1. *Define the valuation σ_d by $n_t = \sigma(n)(t)$ for $0 \leq t \leq d$ Let f be a TEL-formula of depth less or equal to $d' \leq d$ with symbolic variables V, then*

$$\phi, \sigma \models f \Leftrightarrow \phi, \sigma_d \models [f]$$

A direct consequence of this is the following lemma:

Lemma 2. *For all inputs i in I, outputs o in O, states s, s' in S, S', it holds that:*

$$F(i, s) = (o, s') \Rightarrow i, o, s, s' \models [[c]].$$

Proof. (Sketch) Suppose $F(i, s) = (o, s')$ for some valuation i, o, s, s' of the set of nodes $\mathcal{N}$. Consider a definition of a node reference (say m_1), the value of this node reference is given by the application of the *and* operator on the two operands (say m_2 and m_3). This assignment of m_1, m_2 and m_3 models the constraint derived from the definition.

Using the transition constraints, the set of constraints for the behavior of the circuit over d time steps is defined as follows:

Definition 14 (Expanded Transition Constraints). *The* expanded transition constraints *over d time steps, written $[c]$, are defined by, for $0 \leq t \leq d$:*

$$[[c]][n := n_t],$$

and, for $0 \leq t < d$ and $s \in S$:

$$s_{t+1} = s'_t.$$

Every valuation that corresponds to an X-behavior meets the expanded transition constraints.

Lemma 3. *If σ is an X-trajectory, then for every $d \in \mathbb{N}$ the valuation σ_d given by $n_t = \sigma(n)(t)$ for $0 \leq t \leq d$ meets the expanded transition constraints $[c]$*

Proof. By Lemma 2 and induction on d.

From Lemmas 1 and 3 we can directly conclude that the SAT procedure is sound.

Proposition 5 (Soundness). *If all three-valued models of the constraints $[A] \cup [c]$ are also three-valued models of the constraints $[C]$ then the STE assertion $A \Longrightarrow C$ is true in the three valued semantics.*

Proof. Directly from lemmas 1 and 3.

3.5 Other decision procedures for STE

BDD based simulation The standard implementation technique of STE is to replace the signal data type of an existing circuit simulator with BDDs representing ternary values. BDDs can represent ternary values by using the so-called dual rail encoding, in which two boolean values are used to represent a ternary value.

Given a trajectory assertion $A \Longrightarrow C$ of depth d, a simulation is performed over d time steps. At every time step the simulator sets the nodes whose value is specified by the antecedent for that time step, and checks whether the requirements of the consequents hold. As soon as a requirement is not fulfilled the algorithm returns a failure, otherwise it continues until the simulation is completed and returns a success.

Simulator-based STE implementation can only propagate information forwards in time, while many properties naturally need a backwards information flow to be verified. An example of this is when the antecedent contains an assumption about an output which is needed to show something about an input mentioned in the consequent. There exist generalizations of STE which feature algorithms for backwards reasoning [14]. Unfortunately, these algorithms require a fixed point computation, and are more complicated than the original forward STE algorithm.

Intel's Forte system[1] is (as far as we know) the only publicly available implementation of a BDD based STE algorithm. Forte is not sound with respect to Y-semantics of STE; we were able to prove properties that require an information flow from input to output within the same time step, while such properties are not valid in the Y-semantics. Our conjecture is that simulator based decision procedures are sound and complete with respect to the forwards semantics.

BED based simulation with SAT solvers Bjesse et al. [5] have introduced another simulator based STE algorithm. Instead of replacing the signal data type of the circuit simulator by BDDs, they used the non-canonical data structure of Binary Expression Diagrams [3]. Again a dual rail encoding was used to represent ternary values by two boolean values. During simulation they use a SAT solver

to check whether at every time step the symbolic values of the nodes meet the requirements of the consequent. Although their method proved to perform well in finding bugs in an Alpha microprocessor, their approach has the same disadvantage as simulator based STE with BDDs: it only propagates information in a forward fashion. Also BED based simulation is sound and complete with respect to the forwards semantics.

Bounded Model Checking *Bounded Model Checking*[4] (BMC) is a SAT based procedure for checking LTL properties. Given a property, BMC tries to disprove it by considering counter examples of increasing length. Although in theory the method can be used for proving properties (as the maximal length of counterexamples is bounded), in practice it is mainly used for bug finding. BMC can be seen as a sound and complete decision procedure for STE w.r.t. to the behaviour semantics, because given an assertion of length d, the BMC procedure only has to check for counterexamples up to length d.

4 A Three-Valued SAT-Solver

In this section, we show how to adapt a standard two-valued SAT-solver to act as a three-valued solver, by slightly changing its implementation.

4.1 DPLL Style Satisfiability Solvers

Most current-day two-valued SAT-solvers are based on variants of the DPLL algorithm [7, 6, 8, 11]. The state of such a two-valued SAT-solver, for each variable x, keeps track if x has gotten the value 0, the value 1, or has gotten no value yet. In the beginning, no value is assigned any value. The algorithm then interleaves the following two phases:

- **Propagation**: Given the state of the variables, derive for other variables values that they must have in order for there to be a solution at all.
- **Branching**: If propagation cannot find any more values for variables, pick a variable x that has not gotten a value yet, and split the problem in two: one where x has the value 0, and one where x has the value 1. Now, recursively solve the two subproblems.

Whenever a variable assignment is reached where one of the constraints is not satisfied, we reach a contradiction

The DPLL algorithm works with constraints that are clauses, which are disjunctions of literals. Propagation triggers when all literals in a clause but one have gotten the value 0. In this case, the last remaining literal is assigned the value 1, since this is then the only way to satisfy the clause.

4.2 Developing a Three-Valued Solver

Changing our point of view slightly, the state of a two-valued SAT-solver can be seen as a three-valued assignment of values to variables. Here, the value X represents the situation where a variable has not gotten the value 0 or 1 yet.

In the propagation phase, some variables then change their values from X to 0 or 1, but only if they are forced to by a particular clause. Thus, propagation can be seen as computing a three-valued solution that is consistent with a given value assignment.

In the branching phase, a variable x is picked and forced to be either 0 or 1. Thus, we can see the branching phase as the source of two-valuedness of a DPLL-style SAT-solver.

The idea is to adapt a two-valued solver to a three-valued solver by treating the two-valued symbolic variables, which must have the value 0 or 1 in each model, differently from the three-valued node variables, which can also have the value X in a model. We simply restrict the branching phase to only work on symbolic variables! When a propagation phase ends, and a new branching variable x has to be picked, we pick the next symbolic variable that has not gotten a value yet. Moreover, if no such variable exists (i.e. all variables without a value are node variables), we have found a three-valued model of the constraints. The three-valued model can be computed by, for each variable, taking the value it has gotten so far, and X if it has not gotten a value yet.

For each three-valued constraint, we simply generate am identical clause in the SAT-solver we use. When restricting the solver as mentioned above, the clause gets the same meaning as described in 11. The only three-valued models that are not allowed are the ones where all literals are false except for one. This is exactly the case where propagation triggers.

To implement a decision procedure that checks $A \implies C$ for a circuit c, we generate a constraint set containing $[c]$ and $[A]$. We feed these to a SAT-solver that is only allowed to branch on the symbolic variables. When a (three-valued) model is found, we check if it satisfies $[C]$. If so, then everything is fine, and we continue. Otherwise, we have found a counter model to the assertion, and we report the current model.

A more efficient method, the one we actually implement, is to add the negation of the constraints in $[C]$ as an extra constraint. This will lead to more propagation in the SAT-solver, which might be more efficient. Also, when a model of $[c]$, $[A]$, and $\neg[C]$ is found, we know it must be a counter model of the assertion $A \implies C$, so the check for satisfying $[C]$ becomes obsolete. However, we add more propagation possibilities, so some models containing X will disappear. This method will thus find less counter models (it will take away some that contain X in outputs that are mentioned in C), and so it makes more things true.

Since the worse-case time complexity of a SAT-solver is exponential in the number of branches, both implementations have a time complexity that is exponential

in the number of symbolic variables. This is comparable to the STE implementation via BDDs.

5 Benchmarks

In this section we present some preliminary benchmarks of verification runs of STE assertions using our SAT procedure. We presents benchmarks of verification runs of valid properties, but also consider benchmarks for bug finding, where we feed the tool with an invalid property, and the tool has to present a counter example.

The circuits we use in the benchmarks are tree-memories which are shaped like a perfect binary tree. The circuits are designed by ourselves. Of course, it would be very much more interesting to test our tool on real world industrial examples, but up to now we have been unable to obtain such examples.

We have checked a simple STE property for a number of circuits with different address widths and data widths, both for correct and buggy implementations. The buggy implementation contains two bugs: the first and last memory location are not writeable. The assertion we check is:

$$
\begin{array}{ll}
 & wr \textbf{ is } 1 \quad \textbf{and } rd \textbf{ is } 0 \textbf{ and } addr \textbf{ is } A \textbf{ and } in \textbf{ is } V \textbf{ and} \\
\mathbf{N}(& wr \textbf{ is } 0 \quad \textbf{and } rd \textbf{ is } 1 \textbf{ and } addr \textbf{ is } A \ ) \\
 & \Longrightarrow \\
\mathbf{N}(& out \textbf{ is } V \ )
\end{array}
$$

The assertion states that if we write a value to an address, reading the address at the next moment in time should yield the same value.

We want to stress that the benchmarks are all run on an unoptimized, first prototype of tool. For instance, in the prototype the splitting limitation (as described in the previous section) is not implemented; we allow the SAT-solver to branch on all variables.

The reason we have not used the splitting restriction is that we discovered that a naive implementation of the splitting restriction yields worse performance than unrestricted splitting. The reason for this decrease in performance is quite obvious: Modern SAT-solvers contain sophisticated heuristics for deciding which variables to split on. SAT problems for verification of large circuits contain typically hundreds of thousands of variables, of which only very few (say less than a hundred) correspond directly to an symbolic variable. So, by limiting the SAT solver to only split on the symbolic variables we completely cripple its ability to make smart choices for the variables to split on.

A more sophisticated way of implementing the splitting restriction, would be to allow the SAT solver to not only split on the symbolic variables, but also on all variables whose value directly depends on the values of the symbolic variables. Implementing the SAT procedure in this way would give the SAT solver much more freedom in the choice of splitting variables, but would not

increase the number of assertions it can prove. Unfortunately, we were not able to run benchmarks on such a procedure before the deadline of these proceedings.

We compare our results against Forte [1], Intel's in-house verification system, which includes an implementation of STE. The benchmarks show that our first prototype is not able to compete with Forte yet when it comes to proving STE assertions. The bug finding performance (for this particular circuit, bug, and assertion) seems comperable, which is very encouraging.

Tree-Memory Bug Finding			
aw	dw	STE SAT	Forte
10	8	0.55	1.4
11	8	3.7	5.3
12	8	8.3	16
13	8	45	42
14	8	59	110

Tree-Memory Verification			
aw	dw	STE SAT	Forte
8	4	0.7	0.6
9	4	1.6	0.6
10	4	5.7	0.6
11	4	23	0.9
12	4	210	4.5

All times in seconds. All benchmarks were run on a PC with an Intel Pentium IV processor running at 3GHz and 2GB memory under Red Hat Linux. The abbreviations aw and dw denote respectively the address width and the data with of the memory circuit.

Fig. 3. Benchmarks

6 Conclusions

We have given a hierarchy of semantics for STE assertions, and decision procedures for these semantics, one of which is a decision procedure for STE assertions that can deal with backwards assertions. The procedure can be implemented in a simple way, and does not require the use of fixed points as simulator based methods for STE.

Our algorithm uses a three-valued SAT solver. Unfortunately, the naive implementation of such a solver behaves not as well as we had hoped. The performance of modern SAT solvers depends very much on their ability to make smart choices for the variable to split on. Simply limiting the SAT solver to only split on the symbolic variables reduces the performance of SAT solvers significantly, because the number of symbolic variables in the SAT problem is relatively small. In the future work section we describe ideas we have to implement the splitting limitation in a more efficient way, without changing the proving power of the three valued SAT solver.

Experimental results show that our prototype implementation of our SAT based procedure is not yet able to compete with BDD based algorithms for STE when it comes to proving assertions. The bug finding performance seems comparable, which is encouraging.

7 Future work

We plan to optimize our three valued SAT solver by allowing the SAT solver to branch on more variables, without changing the proving power of the solver. A first step would be to allow the SAT solver to not only split on the symbolic variables, but also on all variables whose value directly depends on the values of the symbolic variables. Implementing the SAT procedure in this way would give the SAT solver much more freedom in the choice of splitting variables, but would not increase the number of assertions it can prove.

Also, we would like to investigate optimisations for properties that do not need a backwards information flow. For such properties it is possible to take away the constraints corresponding to the parts of the circuits that are only fed with X's. This would yield smaller SAT problems, that are hopefully easier to solve.

We believe that STE via SAT can be a good complement to BDD-based STE. In order to investigate this claim, we would like to study real world examples that are hard for BDD-based STE. We would like to cooperate with industry to obtain such examples.

We would like to investigate whether our ideas can be generalised to GSTE. One way to do so is to apply our procedure to paths of finite length of an GSTE graph, which can be seen as STE assertions. This would yield a bugfinding method for GSTE. The idea to use STE methods for GSTE bug finding is introduced in [12], where simulator based SAT methods are applied to finite paths of an GSTE graph. Another approach would be to try to translate (subclasses of) GSTE graphs directly to a SAT problem, so that we can use SAT solvers to prove GSTE assertions.

References

1. FORTE. http://www.intel.com/software/products/opensource/tools1/verification.
2. Mark Aagaard, Robert B. Jones, Thomas F. Melham, John W. O'Leary, and Carl-Johan H. Seger. A methodology for large-scale hardware verification. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 263–282. Springer-Verlag, 2000.
3. Andersen and Hulgaard. Boolean expression diagrams. In *LICS: IEEE Symposium on Logic in Computer Science*, 1997.
4. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.
5. P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an Alpha microprocessor using satisfiability solvers. In *Proceedings of the 13th International Conference of Computer-Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 454–464. Springer-Verlag, 2001.
6. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.
7. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *JACM*, 7(3):201–215, 1960.

8. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.
9. Tom Schubert. High level formal verification of next-generation microprocessors. In *Proceedings of the 40th conference on Design automation*, pages 1–6. ACM Press, 2003.
10. Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design: An International Journal*, 6(2):147–189, March 1995.
11. Niklas Eén & Niklas Sörensson. An extensible sat-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, 2003.
12. Jin Yang, Rami Gil, and Eli Singerman. satGSTE: Combining the abstraction of GSTE with the capacity of a SAT solver. In *Designing Correct Circuits (DCC'04)*, 2004.
13. Jin Yang and Amit Goel. GSTE through a case study. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 534–541. ACM Press, 2002.
14. Jin Yang and C.-J. H. Seger. Introduction to generalized symbolic trajectory evaluation. In *IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD '01)*, pages 360–367, Washington - Brussels - Tokyo, September 2001. IEEE.

Verification of Parametric Timed Circuits using Octahedra

Robert Clarisó and Jordi Cortadella

Department of Software
Universitat Politècnica de Catalunya
Campus Nord, Jordi Girona 1-3
08026 Barcelona, Spain

Abstract. This paper presents an approach for the verification of timed circuits in which all delays are symbolic. Abstract interpretation is used to face the complexity of the problem and a new abstraction is proposed to represent timed regions: octahedra. A new type of decision diagrams are used to symbolically represent and manipulate state spaces with octahedral timed regions. The approach has been successfully applied to the verification of asynchronous controllers.

1 Introduction

The correctness of timed circuits depends on the delays of its gates, wires and environment. The conventional approaches for verification rely on the fact that the delays are known a priori, or belong to a constant interval (bi-bounded delays) [4, 16], using behavioral models as timed transition systems [14], timed Petri nets [20] or timed automata [1]. However, the results obtained for a specific set of delay values cannot be extrapolated, in general, to other sets of values.

In this paper we are interested in verifying circuits in which delays are symbolic. As an example, let us analyze the D flip-flop shown in Fig. 1. We want to derive a set of sufficient constraints that guarantee the following property to hold: "*The value of Q after a delay $T_{CK\rightarrow Q}$ from CK's rising edge must be equal to the value of D at CK's rising edge*". Any behavior not fulfilling this property is considered to be a failure.

Figure 1(c) reports a set of sufficient linear constraints derived by the algorithm proposed in this paper. The symbols d_i and D_i represent the minimum and maximum delays of gate g_i, respectively. The most interesting aspect of this characterization is that it is *technology independent*.

There have been few proposals towards the analysis of circuits with symbolic delays. Parametric *difference bound matrices* (DBMs) [3, 15] and *Presburger arithmetic* [2] can handle symbolic delays in the verification of timed automata and timing diagrams, respectively. The incorporation of symbols in verification involves a high computational cost. For example, the approaches for timed automata have been able to handle 5 symbols at most with parametric DBMs.

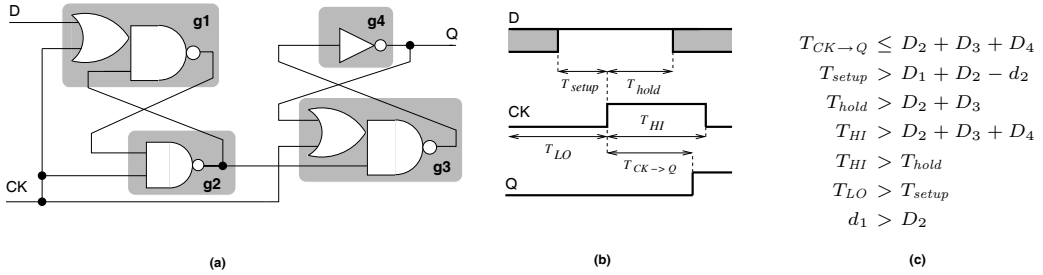
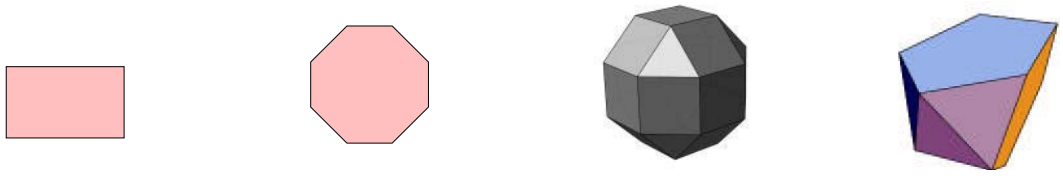


Fig. 1. (a) Implementation of a D flip-flop [19], (b) description of variables that characterize any D flip-flop and (c) sufficient constraints for correctness for any delay of the gates.

Abstraction	Intervals	DBMs	Octagons	Octahedra	Polyhedra
Constraints	$a_i x_i \leq k$	$x_i - x_j \leq k$	$a_i x_i + a_j x_j \leq k$	$\sum_{i=1}^{n} a_i x_i \leq k$	
Coefficients	$a_i \in \{-1, 1\}$		$a_i, a_j \in \{-1, 0, 1\}$	$a_i \in \{-1, 0, 1\}$	$a_i \in \mathbb{Q}$
Example	$-3 \leq x \leq 5$ $0 \leq y \leq 2$	$-3 \leq x \leq 5$ $x - y \leq 4$	$x + y \leq 8$ $y - z \leq 3$	$x + y - z \leq 4$ $x + z \leq 9$	$2x - y + 3z \leq 8$ $2y - z \leq 5$

Table 1. Constraints used for the representation of timed regions ($k \in \mathbb{Q}$).



(a) Intervals (b) Octagons (c) Octahedra (d) Polyhedra

Fig. 2. Different abstractions used in abstract interpretation.

1.1 Abstract interpretation

We resort to abstract interpretation [9] to verify timed circuits with symbolic delays. Abstract interpretation uses approximations of the state space, as a compromise between accuracy and efficiency. The method proposed in this paper is conservative (no false positives) for the verification of safety properties.

When the delays of the components are constant, the timed regions can be represented as conjunctions of inequations of the form $x_i - x_j \leq k$, where x_i and x_j represent the clocks of timed components. The constant k accumulates the delay information obtained from the pre-history of the system in a particular state. DBMs have been typically used to represent these regions. When delays are symbolic, k turns to be a linear combination of variables and cannot be represented as a constant.

Table 1 summarizes the type of constraints used in different abstractions for the approximation of state spaces in abstract interpretation. A set of states is modeled as a conjunction of constraints. The abstractions are ordered, from left (intervals) to right (polyhedra), by their expressive power. Figure 2 depicts some of the regions that can be modeled by the previous abstractions. Intervals can only model regions with one-dimensional constraints. DBMs and octagons use two-dimensional constraints (DBMs is a subclass of octagons in which one-dimensional constraints can be defined by using a variable x_0 with constant value 0). Finally, convex polyhedra use n-dimensional constraints.

In this paper we will use octahedra, a subclass of convex polyhedra in which the coefficients belong to the set $\{-1, 0, 1\}$.

1.2 Why octahedra[1]?

In a previous approach, convex polyhedra were used to verify timed circuits with abstract interpretation [8]. After numerous experiments, one immediately realizes that the required constraints to guarantee most of the circuit correctness belong to the class of octahedra shown in Table 1. Intuitively, they correspond to constraints of the type

$$\underbrace{(\delta_1 + \cdots + \delta_i)}_{\text{delay(path}_1\text{)}} - \underbrace{(\delta_{i+1} + \cdots + \delta_n)}_{\text{delay(path}_2\text{)}} \geq k$$

[1] For simplicity, we use the name *octahedra* to denote a more general class of n-dimensional polyhedra. Among the three-dimensional convex polyhedra belonging to this class we have the octahedron, cube, truncated cube, cuboctahedron, truncated octahedron and rhombicuboctahedron (shown in Fig. 2(c)).

that indicate that certain paths must be longer than other paths. In very rare occasions, coefficients different from ± 1 are necessary. A typical counterexample would be the case in which one path in the circuit must be longer than c times another path (e.g. a fast counter).

The approach presented in this paper is suitable for the verification of small controllers, typically designed by hand or by sophisticated synthesis tools, whose behavior depends on the timing characteristics of the components, such as asynchronous controllers (e.g. [19, 21, 23]). But the technique is also applicable to any level of granularity. For example, one could verify RTL specifications with delays at the level of functional blocks (ALUs, counters, controllers, etc).

2 Timing reachability algorithm

2.1 Overview

The timed behavior of an asynchronous controller can be modelled by viewing its state space as a *timed transition system* (TTS). Each state corresponds to an assignment of values to the signals of the circuit, and each transition corresponds to a change in the value of one signal. Each transition is annotated with an *event*, that defines the component (e.g. gate, transistor, environment) that changes the state of the signal. Events of a TTS can only be fired if their lower and upper bound restrictions $[d, D]$ are satisfied. These delays define the processing time required by a gate to change the value of its output signal after a change in the inputs.

An event can become enabled in one state and be fired in a later state after being enabled for some time. Intuitively, each event has an associated event clock that stores the amount of time elapsed since the transition became enabled. Each time an event is fired, event clocks are updated accordingly. Analysis of the values of event clocks can reveal whether an event can be fired or not in a given state.

The algorithm presented in this section computes a conservative upper approximation of the event clock values, that can be a set of *convex polyhedra* or *octahedra*. Approximations will be propagated and combined using fixpoint techniques described in abstract interpretation [9]. The following sections describe the different parts of the algorithm: the abstract interpretation techniques (2.2); the operations on octahedra (2.3); and the function that updates the clock values after firing an event (2.4).

2.2 Abstract interpretation

Abstract interpretation [9, 10] is a framework of approximate static analysis techniques which can be applied to many kinds of analysis problems in different types of systems. In order to solve a specific problem, the framework of abstract interpretation has to be adapted to:

- *the properties being studied:* The state of a system may contain information which is not necessary to check a given property. Therefore, in our analysis we can work with an *abstraction*, a simplification of the state that ignores the irrelevant information.
- *the semantics of the system:* The behavior of a system can be defined by identifying a set of *locations* where we require information about the state. The relations among the state of the system in these locations establishes a *system of equations*.

The system of equations is solved iteratively using fixpoint techniques, yielding an abstraction that describes an upper approximation of the state in each location of the system.

For the problem of timing analysis of a TTS, a configuration is a set of valid assignments of constant values to clocks and symbolic delays. We will abstract the set of valid assignments as an octahedron that is an upper approximation of this set, i.e. all valid assignments are included in the octahedron. The octahedron will describe the linear constraints that are satisfied among clock values and symbolic delays in all these valid assignments. The locations of interest of our timing analysis

```
Algorithm AbstractInterpretation (R, Inv)
Input: A timed transition system R = ⟨⟨S, Σ, T, s_in⟩, d, D⟩ with an invariant Inv defining constraints on symbolic delays.
Output: The abstraction Time for all states.

  foreach state s ∈ S do Time(s) := ∅; endfor
  Time(s_in) := Inv;
  changed := {s_in};
  do
    n := state in changed with lowest DFS number;
    changed := changed \ {n};
    foreach transition n --e--> m ∈ T
      newTime := transfer(n, e, m);
      if (newTime ⊆ Time(m)) continue;
      newTime := newTime ∪ Time(m);
      if (e is a back edge)
        Time(m) := (Time(m) ∇ newTime) ∩ Inv;
      else
        Time(m) := newTime ∩ Inv;
      changed := changed ∪ {m};
  while (changed ≠ ∅);
```

Fig. 3. Abstract interpretation algorithm

of will be the states of the TTS. We will note the abstraction in a given state s as $\mathsf{Time}(s)$. This abstraction describes the values of clocks when a state is reached, i.e. the *precondition* of the state.

In order to define the timing behavior of the system, we have to build a system of equations that defines how time elapses. When an state is reached, several events become enabled while other events that were enabled previously continue to be enabled. These events have to be fired according to its lower and upper delay bound, taking into account that some events have already been enabled for some time. We have defined a symbolic function called *transfer* (see section 2.4) that advances the clock values while satisfying all upper and lower bounds. The output of this function is the value of clocks after firing an event, i.e. the *postcondition* of the transition being taken. Using this function, the abstractions for states can be defined as the following system of equations:

$$\forall m \in S, n \xrightarrow{e} m \in T \ : \ \mathsf{Time}(m) = \bigcup \mathrm{transfer}(n, e, m)$$

Fig.3 describes an algorithm that computes a solution for this system of equations using a *increasing* fixpoint. Each location starts with an empty set of valid assignments to clocks and values, i.e. an empty abstraction. The algorithm applies the equations iteratively as long as they add new valid assignments. The solution is reached when there is a fixpoint.

Termination, i.e. convergence of the system of equations, is guaranteed by modifying the computation for loops. A *widening* operator [10] is used in the the equations of those states that are the targets of back-edges, i.e edges closing loops. Intuitively, widening extrapolates the effect of iterating a loop an unknown number of times. An in-depth discussion on termination of fixpoints and the necessity of widening can be found in [9, 10].

2.3 Octahedra

Definition 1 (Octahedron). *An octahedron O over $\mathbb{Q}^n$ is the set of solutions to the system of m inequalities $O = \{X | AX \geq B\}$, where $A \in \{-1, 0, +1\}^{m \times n}$ and $B \in \mathbb{Q}^m$.*

The same octahedron can be represented by different sets of constraints. For example, $(x = 3) \wedge (y \geq 5)$ and $(x = 3) \wedge (x + y \geq 8)$ define the same octahedron. Each representation consists of

some inequalities that define several implicit inequalities obtained as linear combinations. In order to consider all the implicit information, all linear combinations of the constraints should be taken into account, keeping the tightest bounds for each constraint. In related data structures such as *difference bound matrices* and *octagons*, efficient algorithms based on the shortest path problem can be used to compute this closure in polynomial time [11, 18]. However, the complexity is polynomial because the number of variables in each constraint is at most two. In the case of octahedra, the closure cannot be computed efficiently. Instead, a weaker closure called *saturation* will be used. Saturation consists on computing all the linear combinations where the coefficients of the linear combination are within the interval $[-1, +1]$. Linear constraints with bigger or smaller coefficients are not considered. The saturation of an octahedron $O = \{X|AX \geq B\}$ can be computed as follows:

1. Select two constraints $A_1X \geq b_1$ and $A_2X \geq b_2$ from $AX \geq B$. Let us define $C = A_1 + A_2$ and $d = b_1 + b_2$.
2. If C contains a coefficient outside $\{-1, 0, +1\}$, return to step 1.
3. If $CX \geq e$ with $e > d$ is already in $AX \geq B$, return to step 1.
4. Add the constraint $CX \geq d$ to $AX \geq B$.
5. Repeat steps 1 – 4 until:
   - A fixpoint is reached (saturation) *or*
   - A constraint $0 \geq k$ with $k > 0$ is found. In this case, the octahedron is empty.

Intuitively, saturation computes all linear combinations of pairs of inequalities. These linear combinations are also new constraints that can be combined with previous inequalities, until a fixpoint is reached. For example, the octahedron $(a \geq 3) \wedge (b \geq 0) \wedge (c \geq 0) \wedge (b - c \geq 7) \wedge (a + b \geq 8) \wedge (a + c \geq 6)$ has the following saturated form:

$$(a \geq 3) \wedge (b \geq 7) \wedge (c \geq 0) \wedge (a + b \geq 10) \wedge (a + c \geq 6) \wedge (b + c \geq 7)$$
$$\wedge (b - c \geq 7) \wedge (a + b - c \geq 10) \wedge (a + b + c \geq 13)$$

In this example, saturation has exposed explicitly that $(a + b \geq 10)$. This inequality is the linear combination of $(a \geq 3)$, $(b - c \geq 7)$ and $(c \geq 0)$. Previously, only $(a + b \geq 8)$ was made explicit. Representating all constraints explicitly will allow an efficient and precise implementation for the union and intersection operators.

Definition 2 (Saturated octahedron). *An octahedron $O\{X|AX \geq B\}$ is saturated if any linear combination of two constraints from $AX \geq B$ with coefficients in $\{-1, 0, +1\}$ also appears in $AX \geq B$.*

Notice that some linear inequalities are not considered during saturation. First, linear combinations with coefficients outside $[-1, +1]$ are ignored. For example, in the octahedron $(a+b \geq 3) \wedge (b+c \geq 8)$, the inequality $(a + 2b + c \geq 11)$ would not be considered. Also, saturation ignores valid linear combinations of three or more constraints if each linear combination of pairs of constraints has a coefficient outside $\{-1, 0, +1\}$. For example, in the system of inequalities $(a - b - c + d \geq 3) \wedge (a + b + c + e \geq 7) \wedge (-a - b + c + f \geq 2)$, the linear combination of the three inequalities is the constraint $(a - b + c + d + e + f \geq 12)$. It does not have a coefficient outside $[-1, +1]$, but the combination of any pair of constraints has a $+2$ or -2. The bottom line is that *the saturated form of an octahedron is not canonical*.

The basic operations used in timing analysis and static analysis of programs, e.g. union, intersection or widening, can be defined in the octahedra domain. In the convex polyhedra domain, the implementation of these operations relies on the *double-description method*: a convex polyhedron can be dually described as a system of linear inequalities, or as a system of generators, i.e. a set of vertices and a set of rays. Some operations on convex polyhedra have an efficient implementation in one of

the dual representations, and the conversion between the two representations is possible and reasonably efficient [6]. The interested reader can find the implementation of convex polyhedra operations in [10, 13].

The problem of the double-description method is the size of the representation, which can be exponential with respect to the number of variables. This complexity severely limits the size of the systems that can be analyzed using this method. The aim of octahedra is overcoming this limitation. Therefore, the implementation of the operations on octahedra will not be based on the double-description method, but in saturation.

All operations on octahedra require that the operands are in saturated form to avoid a loss in precision. However, as saturation does not make all the inequalities explicit, some of the operations may lose precision, producing an upper approximation of the exact result. This is not very relevant when octahedra are used in abstract interpretation, because octahedra are already defining an upper approximation of a set of values in $\mathbb{Q}^n$.

- **Intersection ($x \cap y$)**: The intersection of two octahedra has all the constraints that appear in any of the two operands. If a constraint appears in both octahedra with a different constant, e.g. $AX \geq b_1$ and $AX \geq b_2$, the maximum of both constants is used as in $AX \geq max(b_1, b_2)$. For example,

$$(a \geq 3) \cap \begin{pmatrix} a \geq 2 \\ b \geq 6 \\ a + b \geq 8 \end{pmatrix} = \begin{pmatrix} a \geq 3 \\ b \geq 6 \\ a + b \geq 9 \end{pmatrix}$$

- **Union ($x \cup y$)**: The union of two octahedra has all the constraints that appear in both operands simultaneously. If a constraint appears in both octahedra with a different constant, e.g. $AX \geq b_1$ and $AX \geq b_2$, the constraint $AX \geq min(b_1, b_2)$ is chosen instead. For example,

$$\begin{pmatrix} a \geq 0 \\ c \geq 0 \\ a + b \geq 5 \end{pmatrix} \cup \begin{pmatrix} a \geq 0 \\ b \geq 0 \\ a + c \geq 4 \end{pmatrix} = \begin{pmatrix} a \geq 0 \\ a + b \geq 0 \\ a + b + c \geq 4 \end{pmatrix}$$

  The operands are not shown in saturated form for brevity. For instance, the inequality of the result $(a + b + c \geq 4)$ appears because the operands contain the constraints $(a + b + c \geq 5)$ and $(a + b + c \geq 4)$ in saturated form.
- **Test for equality($x = y?$)**: Two octahedra are considered equal if they have the same saturated form.
- **Test of inclusion($x \subseteq y?$)**: The test for inclusion can be rewritten as $(x \cap y) = x?$.
- **Test for emptiness**: An octahedron is empty if the constraint $0 \geq K$, with $K > 0$, appears in the saturated form.
- **Widening ($x \nabla y$)**: The widening operator can be defined exactly as the widening for convex polyhedra: the result is the constraints of x that are satisfied by y. This operator is used to perform an extrapolation of the behavior of a loop, and is therefore an approximate operator.

  Remarkably, if the octahedra used as operands only contain constants 0 and $-\infty$, then the union operator is already a widening. Having an infinite increasing chain is not possible as each variable can only have three possible values for its coefficients.
- **Existential quantification**: The quantification is performed by removing all the constraints where the abstracted variable appears. The result of this operator is saturated.

The implementation of these operations relies on efficient algorithms and data structures for representing and manipulating saturated octahedra. These tools are provided in the section 3.

```
Function transfer(src, e, dst)
Input: An event src --e--> dst.
Output: The postcondition of src --e--> dst.

  P := Time(src);
  P := P ∧ (step ≥ 0);
  P := P ∧ (clock_e + step ≥ d_e);
  P := P ∧ (clock_e + step ≤ D_e);
  foreach event e' ≠ e: e' ∈ E(src)
    P := P ∧ (clock_e' + step ≤ D_e');
  foreach event e' ≠ e: e' ∈ {E(src) ∩ E(dst)}
    P[clock_e' := clock_e' + step];
  foreach event e' ≠ e: e' ∈ E(dst) ∧ e' ∉ E(src)
    P[clock_e' := 0];
  foreach event e' ≠ e: e' ∈ E(src) ∧ e' ∉ E(dst)
    P[clock_e' :=?];
  if (e ∈ E(dst)) P[clock_e := 0];
  else            P[clock_e :=?];
  P[step :=?];
  return P;
```

Fig. 4. Clock transfer function

2.4 The clock transfer function

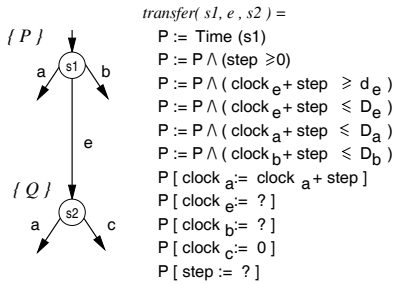
The core of the analysis is the *clock transfer* function that computes *symbolically* the changes in clock values after firing an event. Clock values are represented by an octahedron, with one dimension per event clock and one dimension per symbolic delay. The restrictions of this octahedron represent the restrictions on the clock values in a given state. Intuitively, the purpose of the transfer function is to make sure that whenever an event e is fired, its delay bounds d_e and D_e are taken into account and added to the restrictions on the clock values.

Event clocks for enabled events store the amount of time elapsed since the event became enabled, while disabled clocks are undefined. After firing an event, event clocks should be updated to reflect the time elapsed between the firing of the last event to the firing of current event. This time spent in the state is called clock *step*, and it should satisfy the following properties:

- Step should be ≥ 0, i.e. no negative time increments.
- Step should be long enough to ensure that the firing of e happens at least d_e time units after e was enabled. At the same time, it should be short enough to ensure that e is fired at most D_e time units after becoming enabled.
- Step should be short enough to ensure that any transition that is enabled before firing e is not forced to fire due to its upper bound constraint.

When an event e is fired, the clocks of other events have to be updated. The change in their clocks depends on whether they are enabled or disabled before and after firing e. Events that become newly enabled have their clock reset to zero, while events that become disabled have their clock undefined. If an event remains enabled before and after e, its clock is increased by the clock step. Finally, if an event remains disabled, its clock does not change.

Fig.4 describes the algorithm that computes the transfer function using octahedra operators. Fig.5 shows an example of the computation that would be performed by the algorithm. Events that are enabled before and after firing event e have been increased by an amount in the interval $[d_e, D_e]$, i.e. the unknown clock step. Also, notice that some constraints among the symbolic delays of different events have been discovered. These constraints were imposed over the clock step during the transfer,



$$\{P\} = \{(clock_e = 0) \wedge (clock_a = 0) \wedge (0 \leq clock_b \leq 1)\}$$
$$\{Q\} = \{(clock_c = 0) \wedge (D_e \geq clock_a \geq d_e) \wedge (D_a \geq clock_a \geq d_e) \wedge (d_e + 1 \leq D_b)\}$$

Fig. 5. Example of the transfer function for an event e, with the postcondition Q obtained from a precondition P.

and *implied* several restrictions on the delays that are made explicit when variable step is undefined. For example, the restriction $D_a \geq d_e$ means that event e can be fired only if a is not faster than e. This restriction is implied by the constraints $clock_a + step \leq D_a, clock_e + step \geq d_e$, $clock_a = 0$, $clock_e = 0$.

3 Octahedron Decision Diagrams (OhDD)

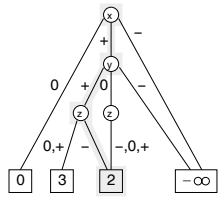
3.1 Overview

In the domain of timed systems, the behavior of a system often depends on clocks and delays. Octahedra describing constraints on these symbols can take advantage of the fact that all variable will be *always* positive. These octahedra can be represented compactly using decision diagram techniques [5]. This representation is called *Octahedron Decision Diagram* (OhDD). Intuitively, it can be described as a Multi-Terminal Zero-Suppressed Ternary Decision Diagram:

- *Ternary*: Each non-terminal node represents a variable v and has three output arcs, labelled as $\{-1, 0, +1\}$. Each arc represents a coefficient of v in a linear constraint.
- *Multi-Terminal* [12]: Terminal nodes can be constants in $\mathbb{R} \cup \{+\infty, -\infty\}$. The semantics of a path σ from the root to a terminal node k is the linear constraint $(c_1 \cdot v_1 + c_2 \cdot v_2 + \ldots + c_n \cdot v_n \geq k)$, where c_i is the coefficient of the arc taken from the variable v_i in the path σ.
- *Zero-Suppressed* [17]: If a variable does not appear in any linear constraint, it also does not appear in the OhDD. This is achieved by using special reduction rules as it is done in Zero-Suppressed Decision Diagrams.

Figure 6 shows an example of a OhDD and the octahedron it represents on the right. The shadowed path highlights one of the constraints of the octahedra, $x + y - z \geq 2$. Notice that all constraints that end in a terminal node with $-\infty$ represent constraints with an unknown bound, such as $x - y \geq -\infty$.

This representation based on decision diagrams provides three main advantages. First, decision diagrams provide many opportunities for reuse. For example, nodes in a OhDD can be shared. Furthermore, different OhDD can share internal nodes, leading to a greater reduction in the memory usage. Second, the reduction rules avoid representing the zero coefficients of the linear inequalities. Finally, symbolic algorithms on OhDD can deal with sets of inequalities instead of one inequality at a time. All these factors combined improve the efficiency of operations with saturated octahedra.



$$
\begin{aligned}
x &\geq 0 \\
y &\geq 0 \\
z &\geq 0 \\
x + y &\geq 3 \\
x - z &\geq 2 \\
x + y - z &\geq 2 \\
x + y + z &\geq 3
\end{aligned}
$$

Fig. 6. An example of a OhDD. On the right, the constraints of the octahedron.

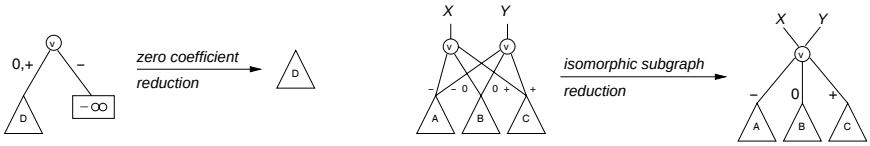


Fig. 7. Reduction rules for OhDD.

3.2 Definitions

Definition 3 (Octahedron Decision Diagram - OhDD). *An Octahedron Decision Diagram is a tuple* (V, G) *where* V *is a finite set of positive real-valued variables, and* $G = (N \cup K, E)$ *is a labeled directed acyclic graph with the following properties. Each node in* K*, the set of* terminal *nodes, is labeled with a constant in* $\mathbb{R} \cup \{+\infty, -\infty\}$*, and has an outdegree of zero. Each node* $n \in N$ *is labeled with a variable* $v(n) \in V$*, and it has three outgoing arcs, labeled* $-$*,* 0 *and* $+$*.*

By defining an order among the variables of the OhDD, we can define the notion of *ordered* OhDD. The intuitive meaning of ordered is the same as in BDDs, that is, in every path from the root to the terminal nodes, the variables of the decision diagram always appear in the same order. For example, the OhDD in Fig. 6 is an ordered OhDD.

Definition 4 (Ordered OhDD). *Let* $\succ$ *be a total order on the variables* V *of a OhDD. The OhDD is ordered if, for any node* $n \in N$*, all of its descendants* $d \in N$ *satisfy* $v(d) \succ v(n)$*.*

In the same way, the notion of a *reduced* OhDD can be introduced. However, the reduction rules will be different to take advantage of the structure of the constraints. In an octahedron, most variables will not appear in all the constraints. Avoiding the representation of these variables with a zero coefficient would improve the efficiency of OhDD. This can be achieved as in ZDDs by using a special reduction rule: whenever target of the $-$ arc of a node n is $-\infty$, and the 0 and $+$ arcs have the same target m, n is reduced as m.

The rationale behind this rule is the following. If a constraint $v_1 + \ldots + c_i \cdot v_i + \ldots + v_N \geq k$ holds for $c_i = 0$, it will also hold for $c_i = +1$ as $v_i \geq 0$. However, it is not known if it will hold for $c_i = -1$. This means that in the OhDD, if a variable has coefficient zero in a constraint, it is very likely that it will end up creating a node where the 0 and $+$ arcs have the same target, and the target of the $-$ arc is $-\infty$. By reducing these nodes, the zero coefficient is not represented in the OhDD. Remarkably, using this reduction rule, the set of constraints stating that "any sum of variables is greater or equal to zero" is represented only as the terminal node 0.

Figure 7 shows an example of the two reduction rules. Notice that contrary to BDDs, nodes where all arcs have the same target will not be reduced.

Definition 5 (Reduced OhDD). *A reduced OhDD is an ordered OhDD where none of the following rules can be applied:*

- Reduction of zero coefficients*: Let $n \in N$ be a node with the $-$ arc going to the terminal $-\infty$, and with the arcs 0 and $+$ point to a node m. Replace n by m.*
- Reduction of isomorphic subgraphs*: Let D_1 and D_2 be two isomorphic subgraphs of the OhDD. Merge D_1 and D_2.*

Definition 6 (Saturated OhDD). *A saturated OhDD is a reduced OhDD where the system of constraints represented by the OhDD is a saturated octahedron..*

All the operations on OhDD work on saturated octahedra. The following section will present the implementation of several of these operations.

3.3 Operations

Reduction rules In order to implement the reduction rules, two basic operations should be defined. First, how to obtain the OhDD for the three cofactors (coefficients) of a given variable. And second, how to build a new OhDD from the three cofactors of a given variable. Figure 8 shows the pseudocode for these two procedures, called `DD_GetCofactors` and `DD_CombineCofactors`.

The only remarkable aspect of `DD_GetCofactors` is how to compute the cofactors of a variable that does not appear in a OhDD. If a variable is missing, it means that it has been reduced, and therefore its negative cofactor is $-\infty$ while its positive and zero cofactor are equal to the OhDD.

Regarding `DD_CombineCofactors`, its pseudocode assumes that there is a procedure called `DD_UniqueNode`) that detects isomorphic nodes in the decision diagram, so if an isomorphic node already exists it is returned, otherwise a new one is created instead. This operation is typically provided by decision diagram packages.

```
Function DD_GetCofactors(f, var)
Input: An OhDD f and a variable var that appears in the ordering before any variable of f.
Output: The 3 cofactors of f for variable var, < f⁻, f⁰, f⁺ >.

  if (DD_IsConstant(f) ∨ DD_TopVariable(f) ≠ var)
    < f⁻, f⁰, f⁺ > := < −∞, f, f >;
  else
    < f⁻, f⁰, f⁺ > := < DD_NegArc(f), DD_ZeroArc(f), DD_PosArc(f) >;
  return < f⁻, f⁰, f⁺ >;
```

```
Function DD_CombineCofactors(var, f⁻, f⁰, f⁺)
Input: The three cofactors of a OhDD for a given variable var. Any variable in the cofactors must
appear after var in the ordering.
Output: A OhDD where < f⁻, f⁰, f⁺ > are the three cofactors for variable var.

  if (f⁰ = f⁺ ∧ f⁻ = −∞)
    return f⁰;
  else
    return DD_UniqueNode(var, f⁻, f⁰, f⁺);
```

Fig. 8. Pseudocode for the auxiliary procedures *DD_GetCofactors* and *DD_CombineCofactors*. These procedures implement the reduction rules of OhDD.

```
Function SaturateRecur(f, g)
Input: Two OhDD called f and g.
Output: The OhDD describing the linear combination of f and g, ignoring constraints with a
coefficient outside {−1, 0, +1}.

  /* Terminal cases */
  if (DD_IsConstant(f) ∧ DD_IsConstant(g)) return DD_Sum(f, g);
  if (f = +∞ ∨ g = +∞) return +∞;
  if (f = −∞ ∨ g = −∞) return −∞;

  /* Lookup the result in the cache */
  res := DD_CacheLookup(SaturateRecur, f, g);
  if (res ≠ NULL) return res;
  top := DD_TopVariable(f, g);
  < f^−, f^0, f^+ > := DD_GetCofactors(top, f);
  < g^−, g^0, g^+ > := DD_GetCofactors(top, g);

  /* Recursive calls for top coefficient = 0 */
  call1 := SaturateRecur(f^0, g^0);
  call2 := SaturateRecur(f^+, g^−);
  call3 := SaturateRecur(f^−, g^+);
  res^0 := MaximumRecur(call1, MaximumRecur(call2, call3));

  /* Recursive calls for top coefficient = +1 */
  call4 := SaturateRecur(f^+, g^0);
  call5 := SaturateRecur(f^0, g^+);
  res^+ := MaximumRecur(call4, call5);

  /* Recursive calls for top coefficient = −1 */
  call6 := SaturateRecur(f^−, g^0);
  call7 := SaturateRecur(f^0, g^−);
  res^+ := MaximumRecur(call6, call7);

  /* Combine the cofactors and update the cache */
  res := DD_CombineCofactors(top, res^−, res^0, res^+);
  DD_CacheInsert(SaturateRecur, f, g, res);
  return res;
```

```
Function Saturate(f)
Input: A OhDD f.
Output: The saturation of the OhDD f.

  do
    old := f;
    res := SaturateRecur(f, f);
    f := MaximumRecur(f, res);
  while (f ≠ old)
  return res;
```

Fig. 9. Pseudocode for the procedures *Saturate* and *SaturateRecur*.

```
Function MaximumRecur(f, g)
Input: Two OhDD called f and g.
Output: An OhDD that has, at the bottom of each path from the root to the terminal nodes, the
maximum terminal found in the same path in f and g.

  /* Terminal cases */
  if (f = g) return f;
  if (f = +∞ ∨ g = −∞) return f;
  if (f = −∞ ∨ g = +∞) return g;
  if (DD_IsConstant(f) ∧ DD_IsConstant(g)) return DD_Max(f, g);

  /* Lookup the result in the cache */
  res := DD_CacheLookup(MaximumRecur, f, g);
  if (res ≠ NULL) return res;

  /* Recursive calls for each cofactor */
  top := DD_TopVariable(f, g);
  < f^−, f^0, f^+ > := DD_GetCofactors(top, f);
  < g^−, g^0, g^+ > := DD_GetCofactors(top, g);
  res^− := MaximumRecur(f^−, g^−);
  res^0 := MaximumRecur(f^0, g^0);
  res^+ := MaximumRecur(f^+, g^+);

  /* Combine the cofactors and update the cache */
  res := DD_CombineCofactors(top, res^−, res^0, res^+);
  DD_CacheInsert(MaximumRecur, f, g, res);
  return res;
```

```
Function Intersection(f, g)
Input: Two OhDD called f and g.
Output: The intersection of f and g.

  res := MaximumRecur(f, g);
  return Saturate(res);
```

Fig. 10. Pseudocode for the procedure *Maximum* and *Intersection*.

Saturation The saturation procedure can be implemented symbolically, so instead of choosing two constraints and computing its linear combination, the linear combination of a whole set of constraints is computed in one step. Figure 9 shows the pseudocode that performs this saturation.

The recursive saturation algorithm `SaturateRecur` computes the linear combination of its two parameters. Intuitively, the computation is split according to the top variable of the decision diagram. The only cases relevant to our computation are those where the top variable will have a coefficient in $\{-1, 0, +1\}$. For example, the linear combination will have a coefficient $+1$ if one of the arguments has coefficient 0 and the other has coefficient $+1$. The remaining variables of the linear combination are computed recursively using the same algorithm.

Saturation is performed by computing these linear combinations and adding them to the system of inequalities until a fixpoint is reached. This computation is described in the procedure procedure `Saturate`. Notice that after computing each set of linear combinations, they are added to the OhDD using the maximum operator, which in saturated OhDD corresponds to the intersection.

Other operations The intersection of two octahedra has been defined as the union of the sets of constraints of both octahedra, choosing the maximum constant for those constraints that appear in both octahedra. In a OhDD, constraints that do not appear in an octahedron are represented by the terminal $-\infty$. Therefore, the intersection of OhDD can be implemented by taking the *maximum* of the two arguments for each path between the root and the terminal nodes. The pseudocode that computes this maximum is shown in Fig. 10. Notice that the result of this maximum is not necessarily saturated.

The same concept can be applied to the union of octahedra. The union can be computed as the *minimum* of the two arguments for each path between the root and the terminal nodes.

The implementation of linear assignments is slightly more complex. In order to apply a linear assignment $[v \leftarrow c_1 \cdot v_1 + \ldots + c_n \cdot v_n]$ to a OhDD, a new variable v' is added to the OhDD. The variable v' holds the new value of v after performing the assignment. Then, the constraint $(v' = c_1 \cdot v_1 + \ldots + c_n \cdot v_n)$ is added to the OhDD using intersection. Finally, the constraints where the old value v appears are removed, and v' is renamed to v.

4 Experimental results

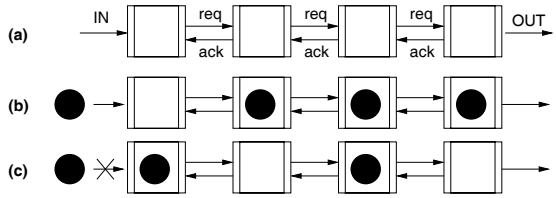
In the initial approach, a set of asynchronous circuits available in the literature were verified using convex polyhedra [8]. These circuits are defined as a network of simple gates plus a STG modeling the behavior of the environment. In these circuits, correctness has been defined as *absence of hazards*, i.e once an event becomes enabled, it does not become disabled before being fired; an *conformance*, i.e. all output events produced by the circuit are expected by the environment. The behavior of the environment is modeled with Signal Transition Graphs (STG) [7]. Table 2 shows the size of the circuits, STGs and the computed TTSs, the number of symbolic delays, the number of constraints required for correctness, and the CPU time used for the verification.

There are two main problems with this approach based on convex polyhedra. First, the discrete state space of the circuit, i.e. all possible combinations of values of the signals, has to be represented explicitly. This limits severely the size of the circuits that can be analyzed, as symbolic techniques cannot be used to represent state spaces compactly. Furthermore, the conversion between the two dual representations of polyhedra sometimes requires a huge amount of memory. In some examples, this conversion requires so much memory that verification cannot be performed. The work on octahedra presented in this paper is an attempt to reduce the memory used by timing verification with convex polyhedra.

The octahedron abstract domain has been implemented on top of the decision diagram library CUDD [22]. In order to implement ternary decision diagrams on top of BDDs, the coefficients of each

Example	Circuit		STG		TTS		# of	# of	CPU Time
	Signals	Gates	Places	Trans	States	Trans	symbols	constraints	(seconds)
nowick	10	7	19	14	60	119	10	2	0.5
gasp-fifo	9	7	10	8	66	209	12	10	8.1
sbuf-read-ctl	13	10	19	16	74	157	14	4	1.2
rcv-setup	9	6	14	15	72	187	12	8	2.1
alloc-outbound	15	11	21	22	82	161	19	3	1.3
ebergen	11	9	16	14	83	188	13	5	1.3
mp-forward-pkt	13	10	24	16	194	574	12	6	1.9
chu133	12	9	17	14	288	1082	7	3	1.3
converta	14	12	16	14	396	1341	14	13	20.4

Table 2. Experimental results using convex polyhedra



# of	State Space		# of	Polyhedra		OhDD	
stages	States	Trans	symbols	CPU Time	Memory	CPU Time	Memory
2	36	88	8	0.6s	64Mb	1.1s	5Mb
3	108	312	10	2s	67Mb	19.9s	8Mb
4	324	1080	12	13.5s	79Mb	291.3s	39Mb
5	972	3672	14	259.2s	147Mb	4825s	57Mb
6	2916	12312	16	–	–	178800s	83Mb

Fig. 11. (a) Asynchronous pipeline with N=4 stages, (b) correct behavior of the pipeline and (c) incorrect behavior. Dots represent data elements. Below, the CPU time and memory required to verify pipelines with different number of stages.

variable v have been encoded using two boolean variables $(v < 0)?$ and $(v > 0)?$. Taking advantage of BDDs, the discrete states of the timed systems have also been encoded symbolically in BDDs.

We have used this library to verify several examples from the domain of asynchronous controllers. The results obtained show that OhDD provide a reduction in the memory usage at the cost of CPU time. This extra CPU time is spent saturating and compacting the OhDD. As an example, we will discuss the case of an asynchronous pipeline with different number of stages and an environment running at a fixed frequency. The processing time required by each stage has different min and max symbolic delays. The safety property being verified in this case was *"the environment will never have to wait before sending new data to the pipeline"*. Fig.11 shows the pipeline, with an example of a correct and incorrect behavior. The tool discovered that correct behavior can be ensured if the following holds:

$$d_{IN} > D_1 \wedge \ldots \wedge d_{IN} > D_N \wedge d_{IN} > D_{OUT}$$

where D_i is the delay of stage i, and d_{IN} and D_{OUT} refer to environment delays. This property is equivalent to:

$$d_{IN} > max(D_1, \ldots, D_N, D_{OUT})$$

Therefore, the pipeline is correct if the environment is slower than the slowest stage of the pipeline. The verification times and memory usage for different lengths of the pipeline can be found in Fig.11. Notice that the memory consumption of OhDD is lower than that of convex polyhedra. This reduction in memory usage is sufficient to verify larger pipelines not verifiable with convex polyhedra.

Surprisingly, an analysis of the results reveals that encoding the discrete state in a BDD-like data structure with OhDD in its leaves has had a negative impact in the execution time. It is very unlikely that discrete state have the same octahedral timing region, even though many constraints hold in several discrete states. As a result, many nodes are shared between OhDD, but the BDD part that encodes the discrete state does not reuse any node at all. Furthermore, dealing with sets of states makes the time *transfer* function more complex, as this function dependes on knowing whether events are enabled/disabled before/after a transition. In an explicit representation of the discrete state, the enabledness of an event can be checked efficiently, while in a symbolic representation it requires additional computations.

References

1. R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. T. Amon, G. Borriello, T. Hu, and J Liu. Symbolic timing verification of timing diagrams using Presburger formulas. In *Proc. ACM/IEEE Design Automation Conference*, pages 226–231, 1997.
3. A. Annichini, E. Asarin, and A. Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In *Proc. Computer Aided Verification*, pages 419–434, 2000.
4. W.J. Belluomini and C.J. Myers. Timed circuit verification using TEL structures. *IEEE Transactions on Computer-Aided Design*, 20(1):129–146, 2001.
5. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
6. N.V. Chernikova. Algoritm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 6(8):282–293, 1964.
7. T.-A. Chu. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, MIT, 1987.
8. R. Clarisó and J. Cortadella. Verification of timed circuits with symbolic delays. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 628–633, 2004.
9. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, New York, 1977.
10. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM Symp. on Principles of Programming Languages*, pages 84–97. ACM Press, New York, 1978.
11. D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, LNCS 407, pages 197–212. Springer-Verlag, 1989.
12. M. Fujita, P. C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, 1997.
13. Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.
14. T. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In *Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 226–251. Springer-Verlag, 1992.
15. T. Hune, J. Romijn, M. Stoelinga, and F.W. Vaandrager. Linear parametric model checking of timed automata. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 189–203, 2001.
16. O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. In *Proc. Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*, pages 189–205. Springer-Verlag, 1995.
17. S. Minato. Zero-supressed BDDs for set manipulation in combinatorial problems. In *Proc. ACM/IEEE Design Automation Conference*, pages 272–277, 1993.
18. A. Miné. The octagon abstract domain. In *Analysis, Slicing and Tranformation (in Working Conference on Reverse Engineering)*, IEEE, pages 310–319. IEEE CS Press, October 2001.
19. C. Piguet et al. Memory element of the master-slave latch type, constructed by CMOS technology. US Patent 5,748,522, 1998.
20. C. Ramchandani. *Performance evaluation of asynchronous concurrent systems by timed Petri nets*. PhD thesis, Massachusetts Institute of Technology, Cambridge, 1973.
21. S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins. Asynchronous Interlocked Pipelined CMOS Circuits Operating at $3.3 - 4.5$GHz. In *IEEE Int. Solid-State Circuits Conf. (ISSCC)*, pages 292–293, February 2000.
22. F. Somenzi. CUDD: Colorado university decision diagram package. Available online at `http://vlsi.colorado.edu/~fabio/CUDD`.
23. Ivan Sutherland and Scott Fairbanks. GasP: A minimal FIFO control. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 46–53. IEEE Computer Society Press, March 2001.

Trading Completeness for Capacity using Probabilistic Techniques

René Krenz Elena Dubrova
Royal Institute of Technology, IMIT/KTH, Stockholm, Sweden

Problem: The growing complexity of verification problems requires new methods that can provide high quality verification coverage for large, complex designs. Conventional simulation is inherently unscalable. It is generally incapable of covering functional corner cases or finding hard-to-find bugs that may occur only after hundreds of thousands of cycles. Existing techniques for an automatic application of formal verification methods have significant capacity limitations and cannot handle large designs in a predictable manner. Our aim is to complement existing simulation-based and formal verification techniques by using probabilistic methods that provide a distinct trade-off between coverage and capacity. Such methods are based on function hashing, allowing us to compute the answer "correct" with a small probability of error, and the answer "incorrect" with 100% certainty. A fast recognition of the "incorrect" answer is particularly useful for finding bugs early in the design flow.

Background: A Boolean function f is hashed by transforming it into an arithmetic polynomial $A[f]$ and evaluating $A[f]$ for a given input assignment of randomly chosen values $\{a_1, \ldots, a_n\}$. If each a_i is an independent and uniformly at random applied value from a prime integer field, then $A[f]$ is the hash value for f, used for probabilistic comparison of Boolean functions.
The arithmetic transform has a broad range of applications. If the individual a_i's are the independent switching activities at the inputs, represented by values between 0 and 1 from a field of real numbers, then $A[f]$ represents the switching activity of a net implementing f, used in power analysis and optimization. Similarly, if a_i is the uncorrelated probability that input x_i is one, then $A[f]$ is the probability that the corresponding net is one. Signal probability analysis is used to improve the coverage of test generation for biased random simulation.

Novelty: Existing algorithms for function hashing rely on building a global BDD or some derivative of BDDs to decompose the input function into disjoint subfunctions. These approaches are limited by exorbitant memory consumption of decision diagrams. None of the existing algorithms hashes functions directly from the circuit representation, which is the core of our approach.

Approach: We have developed an early prototype of an algorithm for hashing a Boolean function from its circuit representation. Our algorithm partitions the hashing process using the dominator relations of the circuit graph. Similar to the application of cut-points in combinational equivalence checking, the dominators are used to progressively simplify intermediate hashing steps. Experimental results show that the new algorithm outperforms BDD-based algorithms and can handle much larger circuits.

Next steps include improving the algorithm and extending it to sequential verification problems, such as bounded property checking. We further plan to investigate the applications of function hashing to cut-points detection for design partitioning. Hash values will be used to classify all internal nets in the circuit and to identify possible cut-points. Finally, we plan to apply the developed techniques to compute signal probabilities for measuring and controlling the coverage of vector generation for biased random simulation. Here the goal is to compute input assignments that maximize the coverage of specific verification targets.

Trading Completeness for Capacity using Probabilistic Techniques

René Krenz Elena Dubrova
Royal Institute of Technology
IMIT/KTH-Stockholm

Overview

- Probabilistic Equivalence Checking based on Function Hashing
- Algorithms for Hashing of Boolean Functions
- Experimental Results I
- Generalized Dominators in Circuit Graphs
- Experimental Results II
- Conclusion

Probabilistic Methods

- good trade-off between coverage and capacity
- "$incomplete$" methods: no false-negatives
- "$sound$" methods: no false-positives

Probabilistic Equivalence Checking

- obtain integer hash code H by evaluating $A[f]$
  on randomly chosen integer vector $S = \{s_1, ..., s_n\}$
  $|S| = n$, n number of input variables

  $S \subseteq F$, F prime integer field
- if $H_1 \neq H_2 \rightarrow f_1 \neq f_2$
- if $H_1 = H_2 \rightarrow f_1 = f_2$, with small error probability $1 - \epsilon$
- $\epsilon = (\frac{|F|-1}{|F|})^{\frac{n}{k}}$, k number of runs with different S
  Example: $(\frac{2^{16}-1}{2^{16}})^{128} = 0.9981$

[1] Jain, Bitner, Fussell, Abraham, $Probabilistic\ verification\ of\ Boolean\ functions$, 1992

Definition of the Arithmetic Transform

- $A[\neg f] = 1 - A[f]$
- $A[g \wedge f] = A[g] \cdot A[f]$, iff $sup(f) \cap sup(g) = \oslash$
- higher-order exponent suppression in case of overlapping support sets
  $x_i \wedge x_i = x_i \quad \rightarrow \quad A[x_i \wedge x_i] = A[x_i] = s_i$

[2] Blum, Chandra, Wegman,

Equivalence of Free Boolean Graphs can be Decided Prob. in Poly. Time, 1980

Definition of the Arithmetic Transform

Shannon's Expansion:

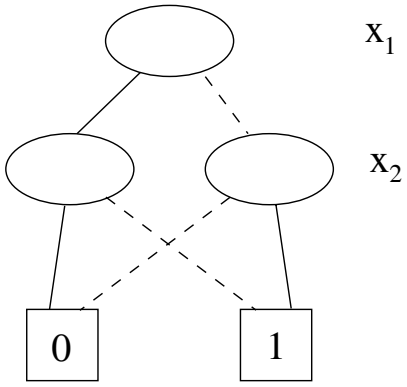
$$f = x_i \wedge f_{x_i=1} \vee \neg x_i \wedge f_{x_i=0}$$

Linear Expansion Theorem:

$$A[f] = s_i \cdot A[f_{x_i=1}] + (1 - s_i) \cdot A[f_{x_i=0}]$$

Evaluating $A[f]$ on BDDs

$$A[x_1 \oplus x_2] = s_1 + s_2 - 2s_1s_2,\ s_1 = 3,\ s_2 = 7$$
$$A[x_1 \oplus x_2] = -32$$



$$A[x_1 \wedge \neg x_2] = -18$$
$$A[\neg x_1 \wedge x_2] = -14$$
$$A[x_1 \wedge \neg x2 \vee \neg x_1 \wedge x_2] = -32$$

Evaluating $A[f]$ on Circuit Graphs

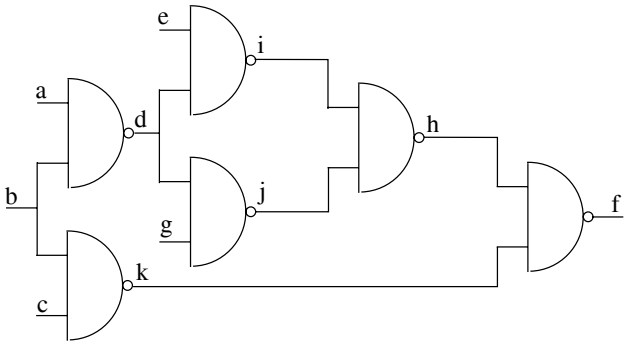
Brute-Force Approach

Arithmetic expression for $A[f]$:

$$A[f] = 1 - (g+e)(1-ab) + eg(1-2ab+a^2b^2) - c((g+e)(1-ab^2) + eg(b-2ab^2+a^2b^3))$$

After suppression of higher-order exponent:

$$A[f] = 1 - (1-ab)(g+e-eg)(1-c) + (1-b)ceg$$



[3] Parker, McCluskey, *Probabilistic treatment of general combinational networks*, 75

Dominator Trees in Circuit Graphs

$$C = (V, E, r)$$

- V, set of vertices
- $E \subseteq G \times G$, connecting edges between gates
- $root \in V$, root node

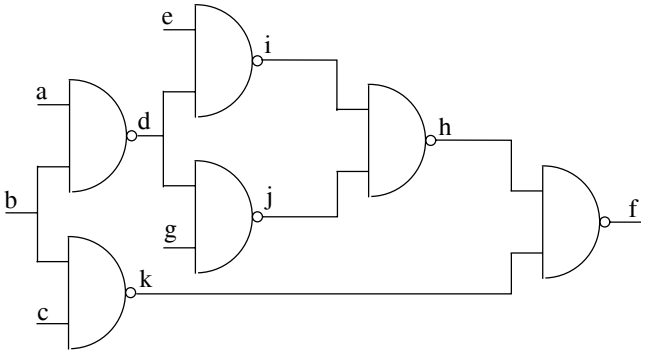
- v **dominates** w $(v = dom(w))$,
  iff every path in C starting from $root$ to w includes v,
  $w \neq v$
- v is **immediate dominator** of w $(v = idom(w))$,
  iff $(v = dom(w)) \wedge (\forall z \in V \mid z = dom(w) \wedge z = dom(v))$

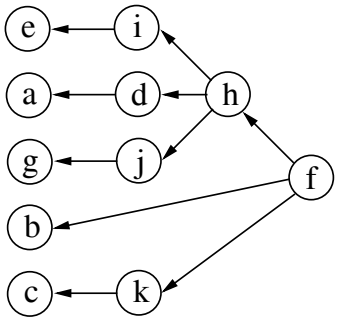
Dominator Trees in Circuit Graphs

Dominator Tree contains all edges
$(idom(w), w) \mid w \in V - \{root\}$
Let $D \subseteq V$ denote the set of vertices of the dominator tree.

Reduced Dominator Tree contains all vertices v such that,

1. v is a primary input; OR
2. $v \in D \wedge \exists u \in D_R \mid v = idom(u)$;





Transformation Algorithm

- immediate evaluation of nets outside of reconverging structures
- introduction of auxiliary variables at multiple fan-out nets
- suppression of these variables at their dominating vertices

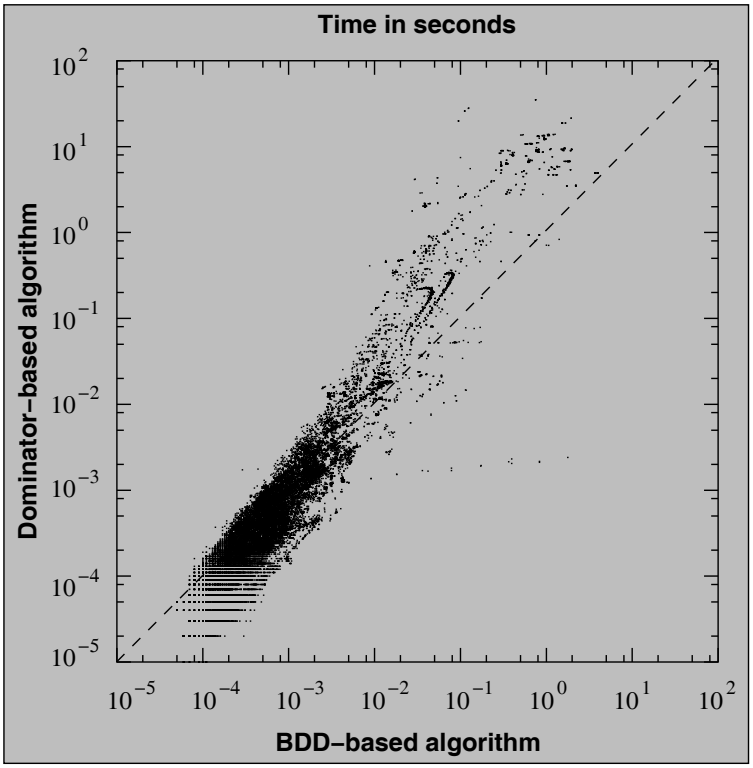
[4] Krenz, Dubrova, Kuehlmann, $Fast\ Algorithm\ for\ Computing\ Spectral\ Transforms\ of\ Boolean\ and\ Multiple-Valued\ Functions\ on\ Circuit\ Representation$, 2003

Transformation Algorithm

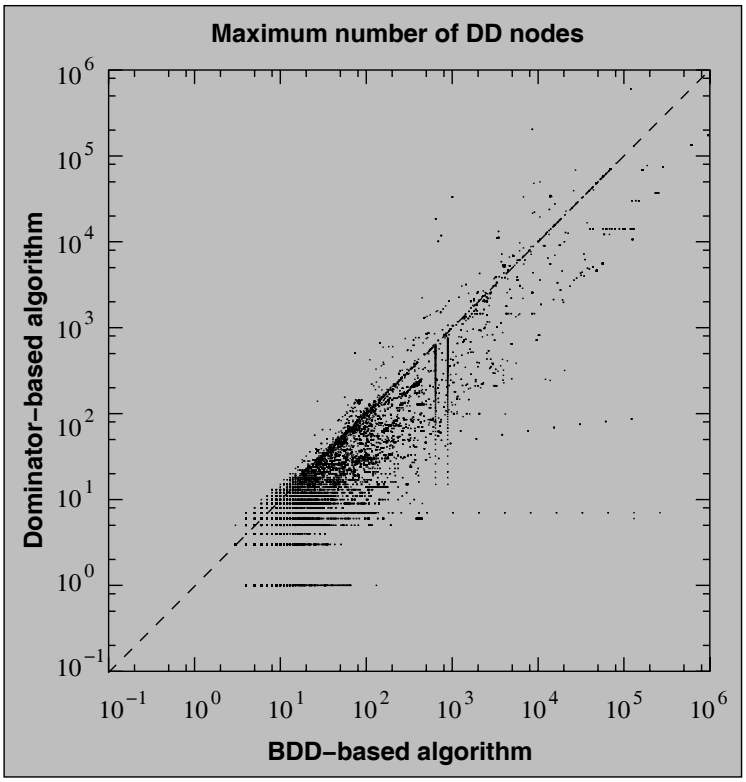
1. computation of single vertex dominators
2. processing of circuit graph in topological order
3. computing $A[v], \forall v \in V$:
   - if v is AND then $A[v] = \prod_{i=fanin(v)}^{i=1} x_i$;
   - if v is NOT then $A[v] = 1 - x_1$;
   - if $v \in Inputs$ then $A[v] = s_i$;
4. substitution of variables created at some $w \mid v = idom(w)$ in reverse topological order
5. create fresh variable iff $v \in D_R \wedge fanout(v) > 1$

[5] Lengauer, Tarjan, $A\ fast\ algorithm\ for\ finding\ dominators\ in\ a\ flowgraph$, 1979

Experimental Results I - Speed



Experimental Results I - Memory



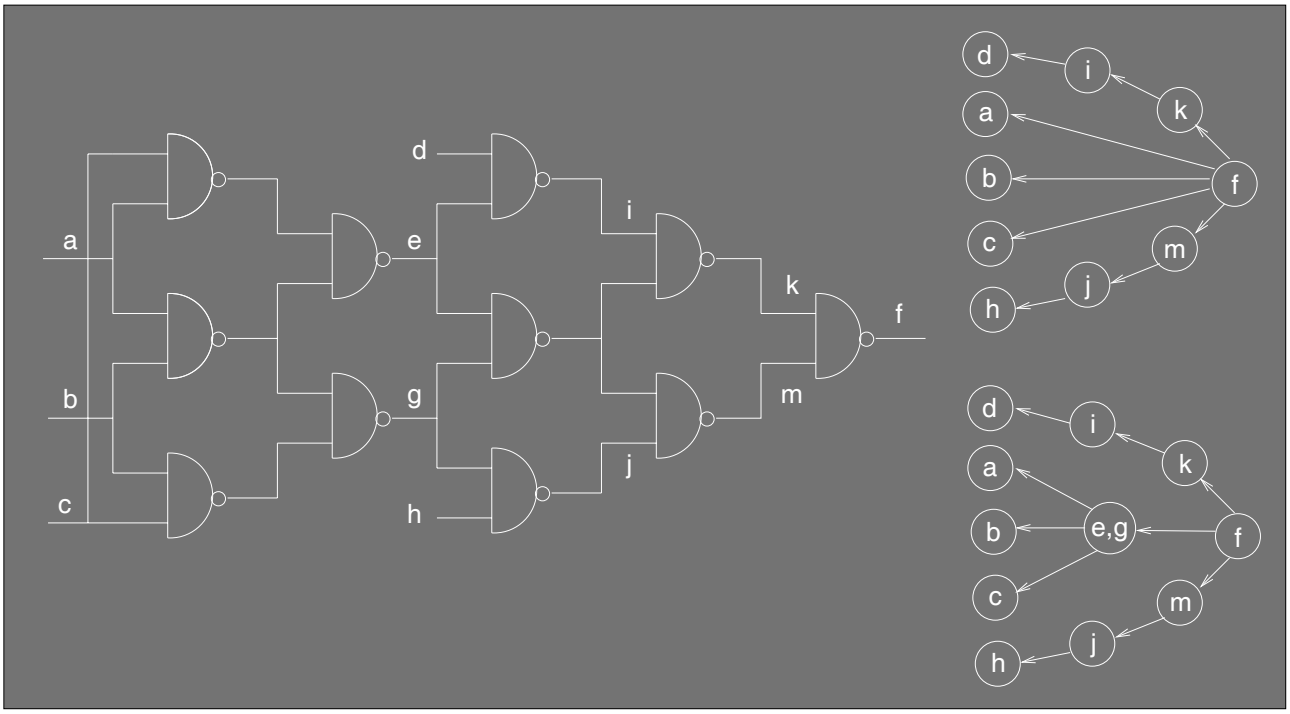
Multiple-Vertex Dominators

A set of vertices $Mdom(u) \subset V$ is a **multiple-vertex dominator** for a vertex $u \in V - Mdom(u)$, if

- every path from u to $root$ contains some $v \in Mdom(u)$;
- for every $v \in Mdom(u)$, there exists a path from u to $root$ which contains v and does not contain any other $w \in Mdom(u)$

[6] Gupta, $Generalized\ dominators\ and\ post-dominators$, 92
[7] Alstrup, Lauridsen, Thorup, $Generalized\ Dominators\ for\ Structured\ Programs$, 00
[8] Alstrup, Clausen, Jorgensen,
$An\ O(|V| * |E|)\ Algorithm\ for\ Finding\ Immediate\ Multiple-Vertex\ Dominators$, 1996

Multiple-Vertex Dominators - Example



Experimental Results II

name	PO	presented algorithm			algorithm [4]	
		$maxN_{DD}$	N_{Mdom}	t_{D_R}, sec	$maxN_{DD}$	N_{PO}
apex5	88	2543	32	0.1139	2626	80
bigkey	421	7557	10	0.2843	7674	224
C432	7	38343	5	0.0075	45114	6
C5315	114	17031	12	0.0378	97640	80
C7552	105	4535	23	0.0470	8082	55
C880	26	1769	4	0.0046	6282	25
cps	109	2422	31	0.0687	3286	93
key	421	3409	224	0.1177	6769	227
s1196	32	1198	13	0.0320	1974	25

Conclusion

- algorithm for evaluation of $A[f]$ on circuit graphs
- processing of all operations on MTBDDs
- usage of dominator relations for analysing re-convergent paths

Formal Verification of Floating-Point Multiply-Add on Itanium® Processor

Anna Slobodova (speaker)

Krishna Nagalla

Massachusetts Microprocessor Design Center

Intel Massachusetts

anna.slobodova@intel.com, krishna.nagalla@intel.com

Motivation

FP instructions work over huge data space that is impossible to cover by means of simulation

Bugs in FP arithmetic are visible and reproducible

High-level specification is "easy" to formalize

- **FP arithmetic well understood and one of the few areas where FV proved to have real impact on the design projects**

It is doable even in the projects of industrial size

- **AMD, IBM, research institutes**
- **Intel has experience of the application of FV to Pentium® processors, but no previous work reported for Itanium® architecture**
- **No previous work reported on the verification of multiply-add**

2

Outline

Verification goal

Specifics of the verification in industrial environment

Tools and methods

Results

Conclusion

3

Verification Goal

Increase confidence in the correctness of the RTL model of floating-point unit of a new Itanium® processor

- **Focus on instructions with big data-space and complex computation (*fma* and related instructions)**

  (Result, exceptions) = normalize (round(normalize(A*B+C)))

  A,B,C are 82-bit Floating-point numbers, 4 rounding modes, 8 precisions, modes for exception handling ~ $2^{81+81+81+2+3}=2^{248}$ possible inputs

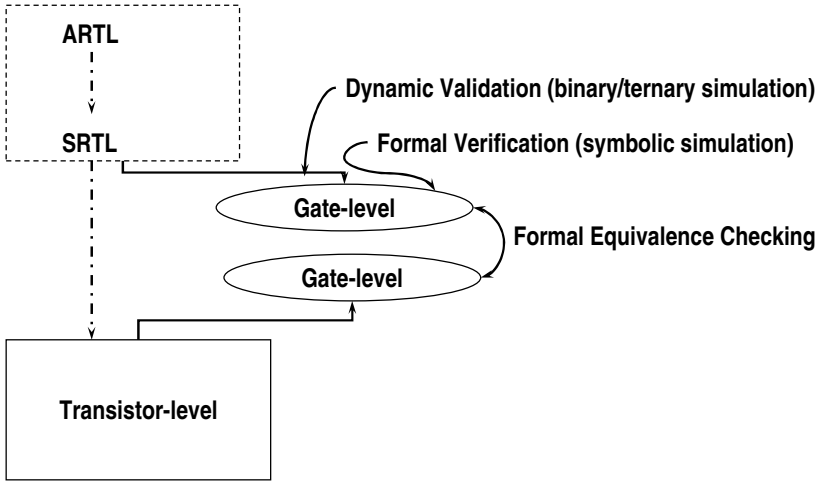
- **Provide full coverage of the computations with normal floating-point numbers**

  X=(sign,exp,mant), sign:{-1,1}, 0 < exp < $1fffe_{16}$, 2^{63} < mant < 2^{64},bias=$ffff_{16}$

  $X=2^{sign} * mant / 2^{63} * 2^{exp-bias}$

4

Verification Goal



5

Specifics of the verification in industrial environment

Incompleteness and continuous changes in the design under verification.

- One of the few FV projects that started in parallel with the design project
- RTL gradually changing from abstract to low-level
- Design often restructured, timing and signal names continuously changing
- New functional and non-functional features added
- Proofs need to be structured for easy management

6

Specifics of the verification in industrial environment

Continuous changes in the specification

- Micro architectural features specified “on the fly” and subject to changes
- Incomplete or not up-to-date documentation

Unstable model requires efficient debugging and replication of the bug in the binary simulation environment

- Extraction of counterexamples that can be reproduced by RTL binary simulation tools

7

Specifics of the verification in industrial environment

Regression of the proofs are required until the completion of the design project

- Creation of the proof is just a start: proofs need to be updated for the changes in the design
- Several thousands lines of FL code, code sharing among different proofs
- Since design is continuously modified, proofs run in a loop

8

Specifics of the verification in industrial environment

- Complexity – design for performance not for verification
  - Hardware is shared by different instructions
  - Scan logic
  - Power management
  - Reset logic
  - Changes do not wait for verification to finish
- Time pressure
- Allocation of resources: Investment / return

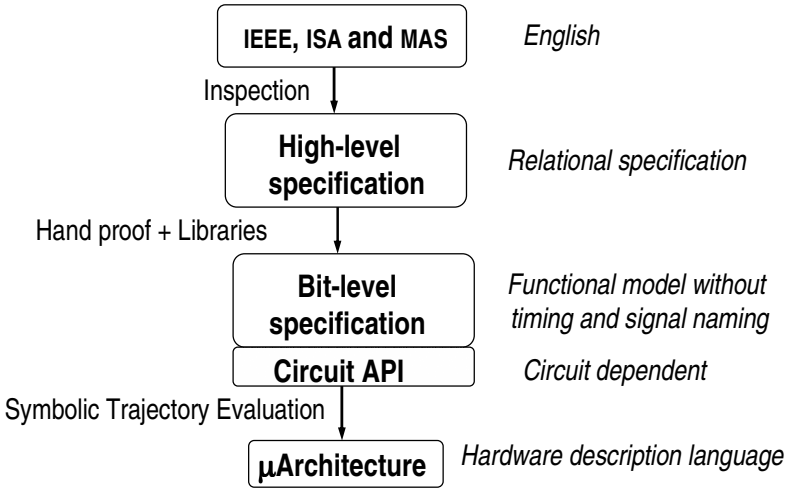
9

Tools and methods

- Based on the methodology applied to Pentium® microprocessor [3,4,5]
- Symbolic Trajectory Evaluation [1] (built-in FL function)
- Propositional reasoning
- Mechanically proved libraries
- Pencil & paper proof decomposition
- No complete mechanical proof
- Maximize gain with minimal resources (re-use what we can)

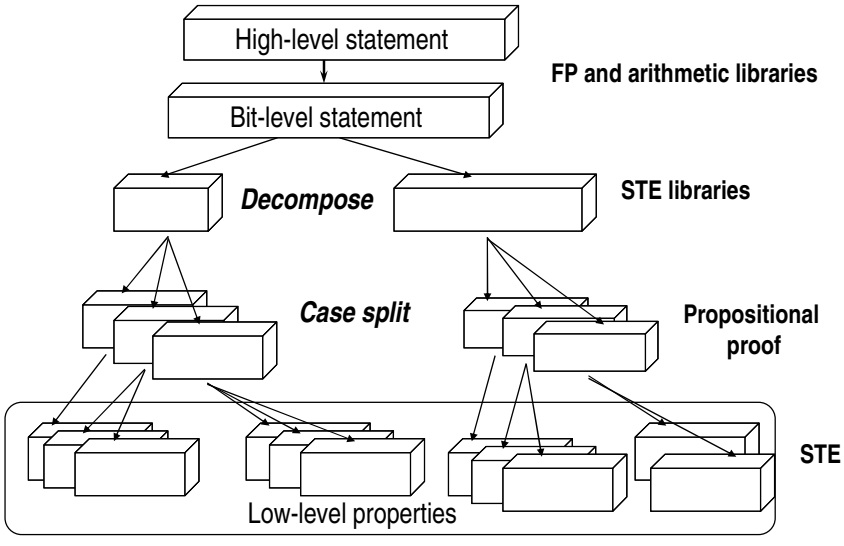
10

Hand-proof + libraries + STE



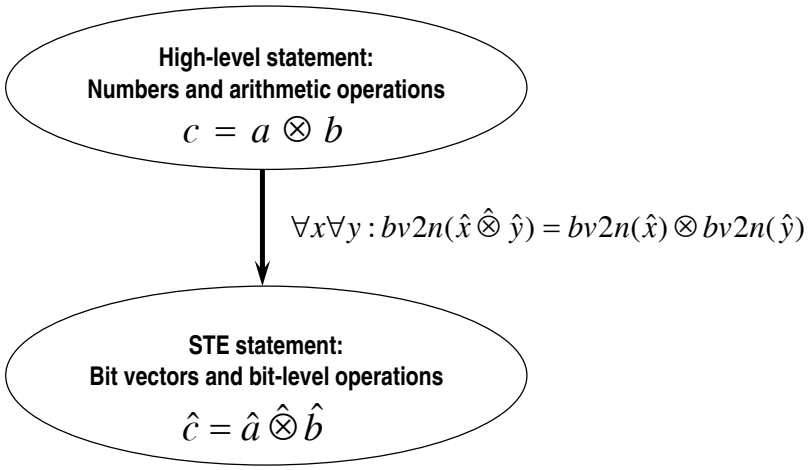
11

Proof structure



12

Arithmetic Libraries



13

Pre-post-condition Framework (Idea)

Used by Kaivola and Aagard [4]

Idea from verification of sequential programs

{P} S {Q} : Precondition *P* guaranties postcondition *Q* after running program *S*

Wrapper around STE

ant: initial state and input stimuli

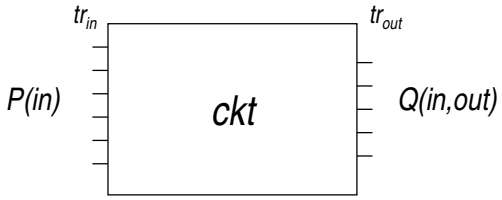
cons: desired response of the circuit

$||=_{ckt}$ [*ant* ==>> *cons*]

All computations of *ckt* that satisfy *ant* also satisfy *cons*

14

Pre-post-condition Framework (STE version)



{ P(in) } (tr_{in}, ckt, tr_{out}) { Q(in,out) }

15

Pre-post-condition Framework (Inference Rules)

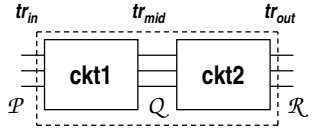
Precondition strengthening

Postcondition weakening

Pre- to postcondition transfer

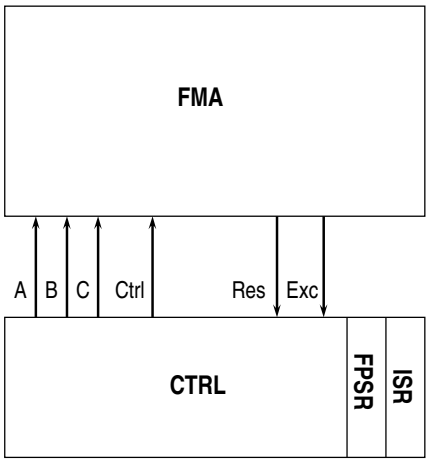
Postcondition conjunction

Sequential composition



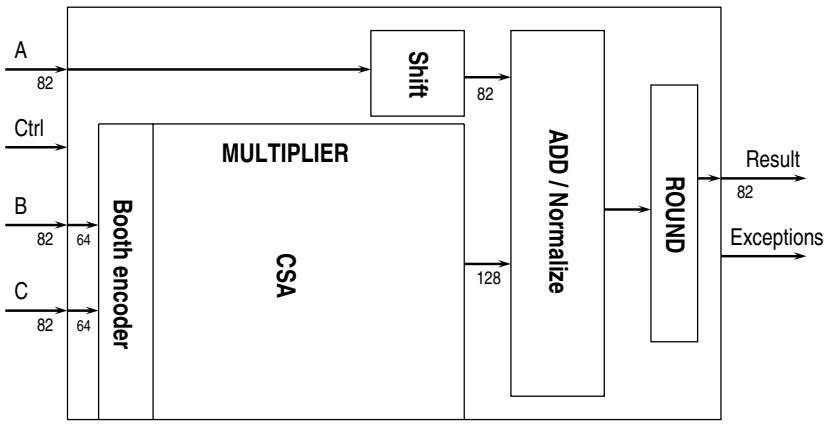
16

Global view of the Design



17

FMA Design



18

Results

Verification of the correctness statement for floating-point and integer multiply-add instruction and other related IA64 instructions - addition, subtraction, normalization, conversion, etc.:

IF

the instruction is started AND

the inputs of the circuit are A, B, C that satisfy input condition AND

control from control circuit is correct AND

expected constraints hold throughout the execution

THEN

At the time the circuit produces output, the output is as specified

19

Results

Proof steps:

- **Correctness of 64x64 Multiplier (follows [5])**
  - **Correctness of the Booth encoder and computation of partial products**
  - **Correctness of CSA (required decomposition)**
- **Correctness of Adder and Rounder (follows [2])**
  - **Alignment shifter required a big case split**
  - **Normalization required a big case split**
  - **Includes exponent and sign data-path, and correctness of the exception flags**

20

Results

Finding bugs
- High quality bugs – hard to reach by random tests
- Bugs invisible to binary simulation
- Bugs masked by other units

Clarifying design interface

Proving redundancy of signals

Benefit of a fast proof for the multiplier

Increasing confidence in the design

21

Results

Many improvements to Micro-architectural specification

Ability to quickly re-run proofs after design modifications

Proofs extended to cover
- Control assumptions
- Special operands and exception handling

22

Conclusion

Verification of multiply-add extended the set of instructions previously verified in industrial projects

- First such effort for Itanium® processor
- Main worry: capacity of the tools

Formal verification of floating-point arithmetic instructions implemented in hardware is recognized as a valuable extension of traditional verification methods at Intel

Huge increase of the confidence in the design correctness with relatively low investment

Initial investment pays-off in the consequent projects

Proof management is important for success

Mechanical proof checking would be a nice closure on our verification

23

Acknowledgement

Our thanks go to

Roope Kaivola and Katherine Kohatsu for helping us understand the methodology and tools applied to the verification of Pentium® processor Floating-point unit;

John O'Leary for encouragement at the early stage of the project;

Our management, architects and validation team for their support of our project

24

References

[1] C.-J.H.Seger, R.e.Bryant: Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Formal Methods in System Design*, 6(2):147-189.1995.

[2] J.O'Leary, X.Zhao, R.Gerth, C.Seger,: Formally Verifying IEEE compliance of Floating-point Hardware. *Intel Technology Journal*, 1999. *developer.intel.com/technology/itj*.

[3] R.Kaivola, K.Kohatsu: Proof Engineering in the Large: Formal Verification of Pentium®4 Floating-Point Divider. *Software Tools for Technology Transfer*, Springer, 2003, Vol.4, issue 3, pp.323-334

[4] R.Kaivola, M.Aagard: Divider Circuit Verification with Model Checking and Theorem Proving. *TPHOLS'2000*. Springer, 2000, LNCS 1869, pp.338-355.

[5] R.Kaivola, N.Narasimhan: Formal Verification of the Pentium®4 Multiplier. *HLDVT'2001.*

[6] A.Flatau, M.Kaufmann, D.Russinoff, E.Smith, R.Sumners: Formal Verification of Microprocessors at AMD. *DCC'2002*.

25

[7] D.Russinoff, A.Flatau: Mechanical Verification of Register-Transfer Logic: A Floating-Point Multiplier. In M.Kaufmann, P.Manolios, J.Moore, Editors, *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Press, 2000.

[8] A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD-K7. Floating-Point Multiplication, Division, and Square-Root Instructions. *LMS Journal of Computation and Mathematics*.

[9] D.Russinoff: A Case Study in Formal Verification of Register-Transfer Logic with ACL2: The Floating-Point Adder of the AMD Athlon Processor. *FMCAD'2000*.

[10] J.Sawada, R.Gamboa: Mechanical Verification of s Square Root Algorithm Using Taylor's Theorem. FMCAD'2002, Springre, LNCS 2517,pp274-291.

26

The Post-Silicon Verification Problem: Designing Limited Observability Checkers for Shared Memory Processors

Ganesh Gopalakrishnan*
School of Computing
University of Utah
Salt Lake City, UT 84112

Ching-Tsun Chou
Intel Corporation
3600 Juliette Lane
Santa Clara, CA 95052

February 25, 2004

Abstract

Today's processors are so complex that static verification methods (simulation and formal verification) alone are insufficient to guarantee correctness. Runtime verification methods help obtain added assurance through the analysis of actual execution traces. Unfortunately, it is becoming much more difficult to access the innards of chips as well as packaged subsystems in order to collect execution traces from hardware. Therefore, methods that minimize access requirements and at the same time can infer a reasonable amount of details pertaining to the internal operation of chips and subsystems are needed. We examine the design of such a *limited observability checker* (LOC) for shared memory multiprocessors in which the processors are interconnected using high-speed links (i.e., no global snooping bus). Our approach assumes that a subset of these links are observable, with the verification problem being one of matching these observations with the designed behavior. We assume that packet traffic on these links can be recorded and presented as a causally consistent sequence for offline analysis by the LOC. After exploring the prototype design of LOCs, we conclude that a special purpose constraint solver constructed using ideas from existing constraint programming languages, symbolic trajectory evaluation, model-checking, and decision procedures holds considerable promise.

1 Introduction

The academic hardware verification community has almost exclusively concentrated on verification, formal or otherwise, conducted at the *pre-silicon* model level, where the models employed include high-level protocol descriptions, RTL descriptions, netlists, and VLSI mask descriptions. Due to the sheer size and complexity of the system under verification, however, there are bound to be logical bugs that escape pre-silicon verification. Thus, it is necessary to conduct *post-silicon* functional verification of both manufactured parts (e.g., microprocessors) as well as fully assembled systems (e.g., microprocessors along with chipsets and peripherals that constitute a shared memory multiprocessor). In this paper, we discuss the growing importance of post-silicon verification for functional correctness, discuss problems that are looming on the horizon, and offer insights into possible solutions based on our initial studies.

In many circles, post-silicon verification means *testing for fabrication faults*. While this form of testing is extremely important, it pays to separate this problem from the problem of detecting logical bugs that have escaped into silicon. We assume that manufactured chips can be subject to tests that reveal the presence of fabrication faults. Such chips can be removed from consideration, and the remaining chips can be assembled into systems (e.g., shared memory multiprocessor) and subject to additional tests to reveal functional (logical) bugs.

*Supported by NSF Grant CCR-0081406 and SRC Contract 1031.001

Why is Post-Silicon Verification Hard?

Today's designs are *wire-centric* [1] in the sense that the amount as well as length of wires dictates product costs as well as performance. Central busses tend to serialize execution. Routing towards a central bus has non-trivial wiring costs that jacks up packaging costs (die size, cooling budgets) as well as the energy necessary to drive long wires. Thus, in highly integrated chips, central observation points that time-order events (e.g., global busses) are avoided. This has the unfortunate effect of reducing overall observability, as many of the point-to-point wires may be located within the integrated circuit die or within packages. Only a limited set of these wires exist outside packages, and hence probed without incurring undue costs. This is the primary reason why post-silicon verification is becoming harder.

The second factor that impacts post-silicon verification is the rate at which information ("new bits") are generated inside chips. In"low-rate" systems where, say, 8-bit microprocessor cores operate at a few 100 MHz, the amount of information generated is, relatively speaking, lower than in systems with much higher data rates comprising, say, four 64-bit CPU cores and three levels of cache, plus directory controller situated inside a single die. These highly integrated high-performance systems are extremely expensive, and extremely reliability-critical. For example, the ASCI-Purple processor slated for deployment at Lawrence Livermore Laboratories will employ 12,000 processors. In such systems, the severity of every bug gets amplified and leads to dramatic reductions in the mean time between failure[1].

Problem Definition

In this paper, we investigate combinations of formal and semi-formal approaches that can make a dent on the post-silicon verification problem. To keep our initial attempts in this area focused, as well as modest in scope, we consider a simple, but real, industrial link-based protocol for cache coherence. For the sake of concreteness, we assume that we are working with a multiprocessor with 4 to 16 CPUs. Each processor 'core' in a multiprocessor introduces a set of correctness requirements that are largely related to its uniprocessor out-of-order execution features. We assume that a separate set of verification methods will be employed for verifying these features. Our work will therefore focus exclusively on multiprocessing aspects. Such partitioning of concerns is inevitable for complexity management. In our description here, we focus on the requirement of *cache coherence* that is included in most shared memory consistency models. We do not consider verification against *shared memory consistency models* [2, 3, 4], leaving it for the future.

We assume that there is support for logging coherence packets on external point-to-point links, and arranging the packet log into a causally consistent history (if packet y is causally dependent on packet x, then x must come before y in the history). We assume that a causally consistent history of some finite length is given to us. We are also given a formal specification of the intended cache coherence protocol at suitable levels of detail. We investigate the construction of an offline analyzer (an automaton) that walks this history, and maintains an estimate of the system state after seeing each packet. The offline checker tool updates the state estimate in the best possible manner upon seeing every new packet. It flags a violation the first time an inconsistency is noted between the internal state and/or an observed packet.

The rest of the paper is organized as follows. In Section 2, we describe an example protocol that highlights some of the difficulties in creating a limited observability checker (LOC). In Section 3, we describe a prototype LOC that was built for an industrial protocol as well as the lessons learnt from that effort. In Section 4, we consider a constraint based approach to building LOCs. In Section 5, we offer our preliminary design for a constraint language, its implementation, and how to build LOCs using this language. Section 6 has our conclusions.

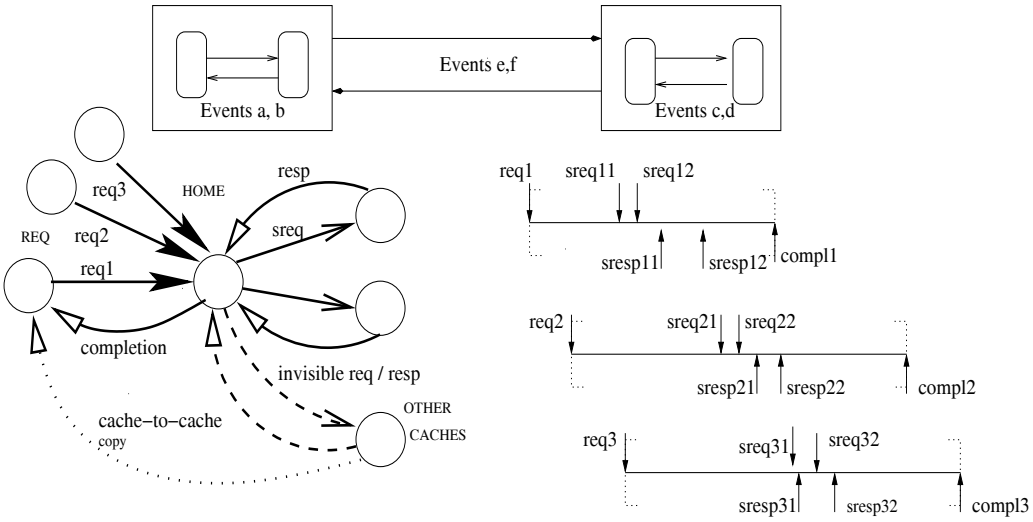


Figure 1: Limited Observability Verification of Multiprocessor Systems

2 An Example Protocol

Figure 1 illustrates a simple cache coherency protocol in operation where events a, b, c, and d are unobservable while e and f are observable. The *communication diagram* shown at the bottom-left of this figure depicts a cache coherence protocol in operation. Assume that the three requesters, req_1, req_2, and req_2, concurrently access same cache line. The requests would typically go to the HOME node (determined as a function of the cache-line address) which selects an arbitration winner, keeping the remaining requests pending. One of the timelines in this diagram also shows the progress of the req_1 transaction. To process this transaction, three snoop requests are sent out by HOME. Assume that only two of these requests, $sreq_{11}$ and $sreq_{12}$, are visible; the third request/response, $sreq_{12}/sresp_{13}$, travel over an invisible link. As soon as $sresp_{12}$ is received, the pending snoop requests $sreq_{21}$ and $sreq_{22}$ are sent out. Proceeding thus, eventually all three requests and responses complete. The third time-line shows a slight temporal variation in that $sreq_{31}$ and $sresp_{31}$ occur before $sreq_{32}$ is sent out. An LOC must be able to process such variations that are equivalent in the same sense as in partial order reduction [5]. The communication diagram also shows the completion of one transaction and a and cache-to-cache copy. Many safety properties are important to check in this example:

- *Data Safety:* At most one other cache may respond with an *sresp* message conveying back the requested data in the exclusive-mode. If this condition is violated, it would mean there exist at least two caches in the exclusive state, which is incorrect.
- *Consistent transaction sequences:* Once a requester receives a cache line in the exclusive state, it must not request a copy for the cache line again until it has explicitly evicted the line or supplied it to another requester. If this condition is violated, it would mean that a node is 'silently' demoting an exclusive line into, say, an invalid or a shared state.
- *Proper Arbitration:* For any address, if more than one *req* packet is received by the home node, only one must win the arbitration. In other words, as soon as the first snoop request (*sreq*) is seen sent out, subsequent *sreq* packets must not be sent out till all the snoop responses are collected by the home node.

Consequences of the Lack of Observability, and Sketch of Solutions

Consider a situation in which we observe two caches supplying the data for the same address in exclusive mode; this is clearly a violation. If, on the other hand, due to observability restrictions,

[1] Jack Dongarra wondered whether the MTBF may equal the time to reboot such a machine!

we can only see one node supplying the data in exclusive mode. Can we then optimistically assume that the node whose response was unobservable is, indeed, correctly behaving (it also is not supplying the same line)? Our approach will be to make such assumptions, but tagging these conclusions *speculative*. They will be turned into the *committed* status as soon as a future *observable* behavior of the same entity (cache line) corroborates the speculative conclusion. Such alternate pathways to observe an entity over different links as well as 'final state' register/memory dumps can be arranged by choosing appropriate tests and by exerting control over the *home node* mapping function. It will be of great interest to find out whether such 'voting' schemes have been studied in the past, and if so under what *error models* they are sound. The idea of 'self consistency' in [6] is the closest work we are aware of. One error model can, for instance, be that an error is not masked by varying the observation pathway.

3 Prototype Implementation of a Limited Observability Checker

Industrial protocols of the nature described are notorious for the the variety of interactions that are possible among the clients, as well as the number of special cases they involve. In order not to be overwhelmed, we assumed that designers would write the tests that generate a meaningful collection of packets on externally observable links. For instance, we assumed that they would pick the addresses employed in the tests in such a manner that under the chosen *address to home node* mappings, and the CPUs involved, a reasonable number of packets would be recorded within each *causal loop*. A causal loop is a sequence of packet-exchanges starting from when a transaction is issued by a processor, ending with the packet that finally conveys the packet to the requester as well as frees-up the home node packet resources for the transaction at hand.

We initially considered approaches that might allow us to *synthesize* the observers, say, by specializing the model descriptions written for a model-checker. For example, one could conceivably take a Murphi [7] model of the protocol and use it as the basis for checking for logical bugs at the post-silicon level. The main drawback of this approach is that finite-state models for complex systems employ many modeling tricks (e.g., the use of non-determinism for over-approximation). It is not immediately apparent how such models may be used for post-silicon verification, given that the recorded packets are generated by actual *deterministic* machines, and not hypothetical machines that employ non-determinism for modeling purposes. Besides, the levels of detail employed at the model-checking level and the actual level are widely different, so a direct carry-over of the code is not possible.

It was then concluded that to be thorough, one must base such a tool on a framework as follows:

- The framework must allow constraints to be added, for example corresponding to the assumptions under which optimistic packet fills are generated.
- When alternatives present themselves, there must be a method for checkpointing the state and later backtracking to a checkpoint. Our initial prototype did not perform backtracking, but was aimed at discovering how much can be done without backtracking.
- There must be support for gathering constraints, quickly calculating the implied constraints, and triggering *intelligent backtracking* to the earliest checkpoint.

A prototype LOC was written in Ocaml for an industrial protocol, and studied on hand-generated scenarios to obtain a feel for the problem's magnitude. In that work, we observed that we could maintain the cache-line states (the traditional MESI states) plus a few auxiliary pieces of information for each cache line. We did record the data associated with the cache lines, however fully realizing that the data recorded within each state element could be of limited value (it is very difficult to monitor the actual data that gets written by the CPU within each cache line; sometimes, the actual data is exchanged through a packet that cannot be recorded).

Briefly, our approach was as follows:

- Catalog the scenarios in which each packet might participate in, based on the coherence message types used
- Develop a *packet attribute table*. For example, for a request packet, the attributes would include
  - the set of all *preconditions* (cache-line states) under which such a request could be issued
  - the immediate post-condition of the cache-line following the issuance of the request packet
  - the number of snoop requests that the directory would generate following a request
  - the directory state-transitions expected following each response packet
  - the state of the requester after it obtains the final response packet
- Since any number of these packets could be unobservable, we had *packet fill* rules for generating the missing packets. This is a delicate issue. Generating the fill packets is based on *optimistic assumptions* about the system (that these packets are *really present* but not recorded simply because they were unobservable). Our approach is to
  - Check that the missing packets are indeed over the unobservable links. If not, generate an error, as a protocol bug has been detected.
  - If the missing packets indeed correspond to unobservable links, we go ahead and generate the missing packets, but also record the optimistic assumptions based on which the fill packets were generated. We call these the *optimistic fill* assumptions. It then becomes necessary to validate the optimistic fill assumptions.

As an example of optimistic packet fills, consider a situation when one of the client nodes has generated a "dirty response"—meaning, it announces that it has the cache line, and will directly supply the line to the original requester. Suppose the response from another client node "x" could not be observed. Under the assumption of coherence, one has to assume that client x does *not* generate a response that signifies that it too has an exclusive copy of the data. This assumption carries an obligation into the future of checking and making sure that client x will not supply the cache line (through, say, a dirty response) over an observable link, in response to a request from another client, unless client x itself has requested and obtained an exclusive copy itself.

3.1 Problems with Initial Prototype Checker

Despite several attempts at development in a clean high-level language, our initial prototype LOC grew in complexity that soon got out of control. The situation-specific handling of packet fillings employed in the code made the code hard to trust. For example, the way we generated fill packets was by calling a dedicated set of routines written for this purpose. This was done to make the code modular, as the same kinds of fill packets were to be generated under different conditions. However, it soon became unclear whether the calling contexts for the packet fill routines were identical or subtle differences were being overlooked. In addition, each such packet-fill routine had certain transaction 'close-out' obligations. In other words, if the final few packets involved in a transaction cannot be observed, they must be inferred, and used to drive the LOC automaton forward to close-out a transaction. If the calling contexts differ, so do the close-out obligations. In summary, we did notice that while an LOC may not have the same kind of complexity as the actual hardware being observed, it does have its own set of corner cases that are indeed *very* insidious. Therefore, there must be a much more declarative and high-level method for expressing the actions of a LOC, and subject the LOC itself to formal verification against higher level criteria pertaining to its consistency and completeness. Thereafter, there must be a way to obtain efficient implementations of the LOCs.

A *constraint-based* organization of the LOC, based on declarative rules, promises to make the code reliable as well as make the handling of invisibility less ad hoc. In fact, we may perhaps not

need to make a priori decisions regarding how much invisibility can be tolerated in causal loops. We can let the underlying constraint system compute solutions based on exactly what is known. We investigate these options in the remainder of this paper.

4 Towards Constraint-based LOCs

We begin with a look at background work in the general area of runtime verification (Section 4.1), and in the area of constraints (Section 4.2).

4.1 General Background and Related Work

Work on online monitoring has been pursued by Havelund and Rosu [8]. In thier work, a past-time based temporal logic is proposed for writing assertions. An efficient dynamic-programming based algorithm is proposed to keep the truths of various subformulas updated upon arrival of the monitored events. Kim et.al. [9] propose a tool for instrumenting Java programs based on a formal specification of the reactive behavior. While there are ideas we will adapt from these works, many of the problems we address are unique because of the limited observability angle pursued. Artho et.al. [10] propose the use of test case generation based on planning. Zhao et.al. [11] employ a model-directed approach to distributed monitoring of hybrid systems that involves a mode-estimation and state tracking algorithm. They apply their approach to monitoring a commercial printer from Xerox. In [12], Bultan introduces a language for interface verification of distributed systems based on ideas from CTL model-checking, constraint solving, and classic work in concurrency based on *path expressions* [13]. Pong and Dubois [14] provide a survey of verification techniques for cache coherence protocols. They do not address the issue of LOC.

Koehler and Treinen [15] present an approach to translate temporal formulae into first-order logic by reification of intervals that leads to an explicit representation of the temporal information contained in the formulae. In our context, this work and others related to it can help us use a constraint formalism to reason about causal sequences of actions not all of which may be observable. Saraswat et.al. [16] investigate the integration of concurrent constraint programming and synchronous programming. Nielsen et.al. [17, 18] study the expressive power of various temporal concurrent constraint programming languages.

4.2 Background Work in Constraints

The paradigm of constraint processing had its origins in Artificial Intelligence [19, 20], Constraint Programming [21, 22, 23], and Formal Verification [24, 25, 26, 27]. Modern Boolean methods based on Binary Decision Diagrams [28], and Boolean satisfiability (SAT) methods [29, 30, 31] are all examples of decidable frameworks for constraint solving. In certain areas, fragments of first order logic are converted into propositional form (e.g., [32]) for decision.

The particular combination of constraint solving techniques brought to bear on a specific problem varies widely. Very few researchers have attempted to bridge the 'constraint programming' point of view and the 'decision procedures' point of view. A notable exception is in the work of Harvey and Stuckey [33] who provide an approach that is superior to the use of *propagators* [21]. The decision procedure of [33] is in fact in use at Microsoft [34] in their device-driver formal verification effort. Several recent efforts have focused on creating stand-alone constraint solving packages for C++ (Modeler++, [35]), Java (JCL [36]), Ocaml (FaCile [37]), and C [38]. Versions of Prolog with constraint programming [39, 40] have been developed. The language Mozart [23] offers several features such as a first-class notion of *constraint spaces* proposed by Muller [41]. In typical applications, the constraint store begins initialized with bindings to variables. Typically the variables are bound integer ranges, but other bindings (e.g., to records) are also possible. Whenever a variable is updated, the associated propagators fire and update the associated variables. Eventually the constraint store stabilizes to a fixed-point. At this stage, if there are still unresolved variables, search begins, with the store being split based on heuristic value assignments

```
(a)

 declare X Y in
  X::90#110
  Y::48#53

 declare A in
  A::0#1000
  A=:X*Y
  {Browse A}  % Displays A{4320#5830}

 X-2*Y:=11
```

```
(b)

Propagation:

S1:  X*Y=:24  X+Y=:10  X=<:Y   ||   X::3#8  Y::3#8

S1:  X*Y=:24  X+Y=:10  X=<:Y   ||   X::3#7  Y::3#7

S1:  X*Y=:24  X+Y=:10  X=<:Y   ||   X::4#6  Y::4#6

Search:

S2: X*Y=:24  X+Y=:10  X=<:Y   ||    X::4    Y::4#6
-->
S2: X*Y=:24  X+Y=:10  X=<:Y   ||    X::4    Y::6


S3: X*Y=:24  X+Y=:10  X=<:Y   ||    X::5#6  Y::4#6
-->
(failed)
```

Figure 2: (a) A Mozart Constraint Example. (b) Details of Propagators and Search

to specific variables. (We are interested in seeing how modern 'intelligent' backtracking methods such as introduced by [30] can be used in this context.) Additional propagator firings result in either a solution or a failed system of constraints. More sophisticated features of Mozart in terms of being able to erect nested scopes of constraint stores, etc, are given in [41].

The object oriented language Comet of van Hentenryck and Michel [42] offers a rich collection of invariant specification mechanisms, the notion of *differentiable constraints* that helps evaluate local moves during search, and the notion of *neighborhoods* that integrates move evaluation and execution in one combined syntactic construct.

4.3 A look at a Modern Constraint Programming Language

The Mozart programming language [23] offers several advanced features in the area of constraints. The features available include the notion of *constraint stores*, the ability to install constraints into the constraint store, and the ability to install *propagators* that keep the constraint store updated. The notion of a constraint space is available as a first-class linguistic primitive.

Figure 2(a) (taken from [23]) presents a simple Mozart program. Variables `X` and `Y` begin as interval-valued variables (shown as `90#110` for `X`, meaning that is the range of `X`). A variable `A` is introduced with initial range `0` through `1000`. Next, a constraint "`A` equal to `X*Y`" is introduced through the syntax `A=:X*Y`. Further constraints can be introduced as shown in the example. Each constraint has the effect of narrowing the intervals associated with the variables. If any interval becomes degenerate, the constraints are not satisfiable. The next example in Figure 2(b) (taken from [23]) attempts to explain what really goes on inside Mozart's runtime.

- The constraint store is initialized with a list of bindings to the variables. Typically the variables are bound integer ranges, but other bindings (e.g., to records) are also possible. There are also an initial set of propagators. In this example, the initial constraint store `S1` contains the three propagators `X*Y:=24`, `X+Y:=10` and `X=<:Y` as shown. It also
- Whenever a variable is updated, the associated propagators fire and update the variable values. In this example, since `X+Y:=10`, we can rule out `X` or `Y` ever being `8`, thus obtaining the second set of bindings, namely `3#7` for `X` and `Y`.
- After a few steps, the constraint store stabilizes to `X::4#6` and `Y::4#6`, and no more propagator firings are possible.

- At this stage, search begins, with the store being split into `S2` (assuming `X::4`) and `S3` (assuming `X::5#6`). Such splitting decisions are taken based on heuristics.
- Additional propagator firings now cause `S2` to evolve to a successful store binding `X::4` and `Y::6`, while `S3` fails.

More sophisticated features of Mozart in terms of being able to erect nested scopes of constraint stores, etc, are given in [23].

4.4 Assessment of Today's Constraint Frameworks for Building LOCs

In this section, we examine some of the reasons why we believe that the constraint solving frameworks available today will not be a ready fit for our needs regarding LOCs.

Handling Multi-domain Constraints

The constraints that LOCs need to represent and solve do not follow a single paradigm such as interval arithmetic. A constraint that captures the details of a single transaction include the following:

- The constraints imposed by a transaction apply to the states of the requesting node, the home node, the nodes that supply the cache line, and the nodes that conflict with the current transaction. The nodes that conflict could be those that conflict for arbitration at the home node. They could also be nodes that conflict when the requesting node for one transaction also assumes the role of a node that supplies the cache line for another transaction.
- Each transaction follows a finite-state protocol. This means that the activities of a transaction affect the various state elements it deals with according to certain causal relationships. Thus, the LOCs must be built based on frameworks that can deal with partial orders of causal dependencies as well as invariants pertaining to the states.

Here is, for example, how two transactions can interact in an industrial protocol (refer to Figure 1 for details):

- Imagine a transaction t_1 that originates at a requester r1 (one of the nodes shown as REQ towards the left of Figure 1). Let this transaction pertain to flushing a cache line which is in the exclusive state.
- Let t_1 be sent to the HOME node which is meanwhile entertaining a transaction from another requester for the same address. The ensuing arbitration conflict causes HOME to send t_1 back to r1 for *reissue*.
- Before r1 can reissue t_1, a cache c1 (one of the nodes shown on the right-hand side of Figure 1 as CACHE) requests the home node for the cache line through another transaction, say c_1. HOME sends c_1 to r1.
- Transaction c_1 finds the transaction t_1 waiting in the outgoing buffer of r1. It essentially "hijacks" transaction t_1 which will no longer get issued.

Now imagine that we can record transaction t_1 going from r1 to HOME, being sent back to r1, and then never emerging again! Assume that due to limited observability we cannot see transaction c_1 going from the cache to HOME - but we can see c_1 going from HOME to r1. How can we generate constraints to fill in all invisible link activities as well as the internal states? We describe a plausible framework in Section 5.

The general question is how to perform constraint solving in the context of reactive nondeterministic protocols? In the constraints literature, extensions of constraint solving into the domain of simpler languages is considered; for example, Saraswat [16] considers constraint solving under the ideal synchrony hypothesis assumption ("Esterel-like" [43]). Extensions to reactive/nondeterministic situations appear to be under intense, but preliminary investigation [18].

Lack of support for datatypes of interest

Constraint programming languages provide many data types on which constraints can be specified. These types could, for instance be interval-valued variables, as well as arrays and records of such variables. For the purposes of building LOCs to monitor shared memory hardware, it is necessary to have variables that denote sets of states, variables that correspond to aggregate structures such as various tables and embedded memory devices, and higher level objects such as partial orders. Most constraint programming languages do not support a rich collection of these data types as well as a uniform way to impart constraints on them.

Propagators of inadequate generality

While the idea of constraint propagators is quite elegant and powerful, the propagators found in constraint languages are of a limited variety. A much more general-purpose propagator can help constrain an entire transaction. What we need in specifying an LOC are propagators that not only tie together the different points in time associated with a transaction, but different aspects of related transactions, such as in the example described in Section 4.4. In the diagram, for instance, we show three request transitions, namely req_1, req_2, and req_3 interacting at the arbiter of a home-node. A constraint must be specified that checks that these transactions share an address, and if so abide by the mutual exclusion requirement of an arbiter. Another constraint pertains to the various snoop responses received, but this time apply to a single transaction. One has to be able to describe that the snoop responses are all concurrent, but only after all of them have been received can the final completion be issued. One must also require that there be at-most one supplier of the value among the snoop responses. Thus, temporal constraints, partial order constraints and constraints on aggregate data must all be expressible in the same propagator. As we describe in Section 5, a propagator in our framework will have the ability to specify a partial order of event observations and the associated data values at all time points of this partial order.

Inefficiency as well as Incompleteness

As pointed out in [41], many constraint solving frameworks either end up exhaustively enumerating the cartesian product of the domains of many variables before realizing that the constraints cannot be satisfied. It is mentioned in [41] that the constraints

$$2x = y \ \wedge \ 2u = v \ \wedge \ y + 1 = v$$

are used to benchmark solvers due to the significant propagation time exhibited by it. Solving using propagators is also incomplete; for instance, using propagators alone, it is impossible to realize that for variables x, y, z in the range $\{0, 1\}$, the constraints

$$x \neq y \ \wedge \ x \neq z \ \wedge \ y \neq z$$

are infeasible. We believe that a constraint framework that retains completeness, even at the risk of forcing behaviors to be expressed in a finite-state manner, will be important to demonstrate useful results achievable through automatic means.

Image Calculations

From the example in Section 4.4, it must be clear that one should have the ability to key off a visible event and trace its precursor event(s). This requires the ability to compute forward as well as backward images of transition relations easily. This again calls for the use of decidable theories, and perhaps modern Boolean reasoning systems that support these operations efficiently.

Lack of modern search features

The search strategies discussed in the context of many constraint systems (e.g., Mozart) do not emphasize the role of intelligent (e.g, non-chronological) backtracking methods. These methods are widely known to be the reason why modern SAT tools (e.g., [30]) and decision procedures (e.g., [27]) perform well. This will be one item studied in the context of our work.

5 A constraint formalism suitable for building LOCs

Let *mid* denote the identifiers for various "modules" (physical units) that exchange the packets during the execution of a cache coherency protocol. These packets are the primary observables for LOCs. Additional observations may be made of the final state (at the end of the test run) of various cache states; we do not specifically address how this extra information might be used to reduce the possible outcomes. Each packet carries many pieces of information that include at least the following for *request packets*.

$$\langle trtype, mid, tid, sid, addr, data \rangle \tag{1}$$

Here, field *trtype* determines the operation being carried out. Field *mid* is the module originating the transaction. Field *tid* is the transaction ID (a unique ID that identifies transactions originating from *this mid*). Field *sid* is the ID of the sender. When a node issues a transaction, both *mid* and *sid* are the same. However, when an intermediate node (such as the home node) forwards the packet, the *mid* and *tid* stay the same, while the *sid* is replaced with the ID of the home.

Request packets are those that correspond to the phase of transactions where the outcome (data returned) has not been determined. In Figure 1, requests flow from requesters to home, or home to other caches. Response packets are generated to send responses back; in Figure 1, they flow from the caches to home, or home to the original requester. In typical coherence protocols, responses that return in the 'downstream direction' on the same links on which requests were sent in the 'upstream' direction may not carry all pieces of the payload: typically they leave the address out, as it can be determined knowing the mid, tid pair that is a unique descriptor for the transaction, once it has been created. We however assume that all packets logically follow the format described in Equation 1—as if the address information is filled in. We however assume that observability is symmetric: if a packet going from m_1 to m_2 is observable, so is any packet going from m_2 to m_1.

It is assumed that, given any packet, the following pieces of information are determined:

- From its *trtype*, we can determine what *communication pattern* the packet will follow. A communication pattern is two pieces of information, $\langle top, prot \rangle$:
  - *top* is a subgraph embedded in the topology of communication shown in Figure 1 showing how the transaction evolves from inception to completion.
  - *prot* is a finite-state protocol specifying how the transaction evolves: for example, it might describe that the transaction first sends a request. Then for every cache the home sends a request and awaits a response concurrently. Then the final response is sent by the home node. We assume that *prot* can be specified in any standard manner, and formally specifies a transition relation. For example, regular expressions augmented with the $\|$ operator to specify finite concurrent threads can be used to specify *prot*.
- From *trtype*, we can also determine the *state* of the transaction with respect to its protocol *prot*. For instance, if we observe a snoop request but not the original request, we can determine that we are in a state of the protocol *prot* where the request has been sent out.

The state of an LOC is a partial function from mid, tid pairs to the following (we use partial functions since many transactions may not have been created yet, or we may not have noted its origin due to limited observability):

- *addr*, the address involved in the current transaction
- *cache*, the state of the participating nodes. We will not assume any particular structure for this state, except to note the features to be included:
  - Instead of saying that a cache is in a specific state (e.g., the exclusive state), we will record what *set of states* it is likely to be in (e.g., {E,S,I}).
  - We will keep various resource bits (e.g., arbitration states of modules)
  - We will have the ability to tag any given state to be *speculative* or *committed*. Speculative states are states inferred by the event propagation rules (to be explained) while committed states are states that might have been speculative once, but are confirmed by a hard evidence (e.g., the precondition of a certain transaction might be that the cache is most definitely in state E).

Semi-formal Sketch of the Constraint Update Algorithm

The LOC state update algorithm will work as follows:

- Obtain the next observable packet, and obtain its fields.
- From its *trtype* field, obtain the protocol *prot* that it is following.
- From mid, tid, see if a mapping is present in the LOC state; if not, inaugurate a transaction inside the LOC state; else (if already present), add the details to be discovered to the LOC state.
- Obtain *prot* as well as the state within *prot* that this transaction corresponds to.
- Perform the event propagation steps to be described next.

Event Propagation, Abstract Interpretation, and Constraint Voting

Many events will have multiple precursors. The resulting nondeterminism in identifying the precursor(s) is handled by propagating the speculative marking to all precursor branches, backwards in time. For each *inferred* state that is a precursor for an observed event, we will assert that the state has a *speculative* status. When two precursor computation branches intersect, we can combine the information as follows:

- Both branches mark the state to be speculative. The set of states marked by the branches are *compatible* (see below). The *resultant* (see below) set of states is computed, and the annotation stays speculative.
- The first branch marked it as committed with state s_1 while the later branch marked it as speculative with state s_2 where s_2 is not compatible with s_1. In this case, the speculation is abandoned, and the precursor path being explored fails.
- The first branch asserts it as speculative in s_1 while the second (later) branch marks it as committed in s_2 which is compatible with s_1. In this case, the resultant is computed, and the annotation changes to the committed status.

As pointed out in Section 2, by controlling test-program parameters, we can ensure that the constraint voting happens many times. In the overall algorithm, if any one speculative precursor branch going back in time succeeds, the current event is admissible, and the LOC state updating proceeds. Of course, 'and' conditions (multiple precursors to have happened) will require all precursors to succeed. If all precursor branches fail, the currently observed event is erroneous, and we flag an error. Note that the two precursors might be independent in the sense that recording them in the packet history in either order is equivalent. We will ensure that the weakest preconditions we choose to associate with each such event ensures that the order of observation does not matter. This will ensure that when we mark a state to be committed while later contradict it with another state that is speculative, the associated failure of the precursor computation is indeed genuine, and not an accident of the observation sequence we happen to pick for the independent events.

Finally, since we are maintaining sets of states, one reasonable approach is to define *compatibility* to be true when the set intersection of the participating sets is non-empty. The corresponding notion of computing the *resultant* set of states is to perform set intersection. We will investigate the semantics of computing the resultant in various ways:

- The resultant computed according to an *intersection* of the speculative state sets corresponds to the *must* semantics (the conclusions are inevitable) while that computed using *union* corresponds to the *may* semantics (the conclusions are possible). What space of possibilities exist?
- What are the connections between the properties verified and the approach taken to define compatibility as well as the resultant? Can we, say, cast these in the light of three-valued temporal logics (e.g., [45])?
- What formal theory of event observations and temporal logics explains the notion of independence identified earlier?
- Last but not least, what are the most efficient methods to implement the above process of limited observability verification in general, and in specific contexts?

Use of Statistical Information

The main steps during event propagation are to find out which precursor events occurred. One could *employ statistical information* to prioritize this search. To the best of our knowledge (*e.g.*, surveying recent papers as well as recent books such as [19]), the use of probabilistic relational models [44] to statistically bias the search conducted within constraint processing systems appears relatively unexplored. Given that very strong event correlations exist in practical systems (*e.g.*, if certain events happen, certain precursors must have happened with a very high probability), and given that conditional probabilities can be calculated from execution trace data (say, from a simulator), it appears a fruitful direction to explore. The mechanism of differential constraints and differential functions offered in [42] might prove to be a very modular way to implement these features. In particular, in [42], the abstract class `Constraint` is extensible with customized methods for move evaluation; some of these new methods could be probability based.

Pragmatic Considerations: Compressing Domains of Observation

Since the packets may carry addresses over a wide range of address values, and since we intend to use finite-state / BDD methods in building an LOC, we assume that there is an injective *remapping* facility that can map the current set of addresses encountered to a 'fictitious,' but contiguous range of values. We also assume a corresponding 'inverse mapping' table to be set up. We assume that this can be done for every attribute in a packet. This way, the packet observations and error reporting can be done with respect to "physical addresses" while the LOC processing happens with respect to "virtual addresses" using efficient finite-domain reasoning methods.

6 Conclusions

In summary, we report preliminary results from our study of the limited observability checking problem applied to industrial cache coherence protocols. After considering several approaches as well as building a few prototypes, an approach that promises to scale to realistic protocols has been articulated. This approach relies on using the constraint technology but in many novel ways. In our proposed approach, we compute precursors to observed events, and mark the states discovered with a *speculative* or *permanent* annotation. We then seek consensus among various annotations that might impinge on a state. The intersecting precursor paths going back into time have the possible outcomes of sharpening speculative states to a narrower set of possibilities or outright contradicting a past state, thus cutting off one branch of precursor computation going into the past. When all branches of precursor computations are cut off, an error has been discovered.

Our remaining work is how to realize a constraint store such as we describe using efficient search methods as well as decision procedures. We hope to define a formal semantics and build a prototype tool by the time of the workshop, and apply it to realistic protocols.

For our implementation, we view the Comet [42] framework with interest: (i) it provides an object-oriented framework for organizing the code; (ii) it provides a rich language for specifying constraints as well as events to listen to; (iii) it provides a checkpointing facility that is based on storing reverse-image computation rules (an idea also used in the SPIN model-checker [46]).

Acknowledgements

The first author wishes to thank Mani Azimi, Mani Ayyer, Ching-Tsun Chou, Jay Jayasimha, Akhilesh Kumar, Phanindra Mannava, Seungjoon Park, Roy Saharoy, Aniruddha Vaidya, and Pascalin Amagbegnon, for discussions and feedbacks during his industrial sabbatical when this work was conducted.

References

[1] Wolfgang Roesner. What is beyond the rtl horizon for microprocessor and system design? Keynote lecture at Charme 2003, L'Aquila, Italy, October 2003.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.

[3] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Analyzing the intel itanium memory ordering rules using logic programming and sat. In *CHARME*, pages 81–95, 2003. LNCS 2860.

[4] A formal specification of intel(r) itanium(r) processor family memory ordering, 2002. http://www.intel.com/design/itanium/downloads/251429.htm.

[5] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, December 1999.

[6] Robert B. Jones, Carl-Johan H. Seger, and David L. Dill. Self consistency checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1166 of *Lecture Notes in Computer Science*, pages 159–171. Springer-Verlag, November 1996.

[7] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design*, pages 522–525, 1992.

[8] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 342–356, April 2002.

[9] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a run-time assurance tool for java programs. In *First Workshop on Run-time Verification*, Electronic Notes in Theoretical Computer Science. Elsevier, July 2001.

[10] Cyrille Artho, Doron Drusinksy, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Grigore Rosu, and Willem Visser. Experiments with test case generation and runtime analysis. In *10th International Workshop on Abstract State Machines (ASM 2003)*, March 2003.

[11] Feng Zhao, Xenofon D. Koutsoukos, Horst W. Haussecker, James Reich, Patrick Cheung, and Claudia Picardi. Distributed monitoring of hybrid systems: A model-directed approach. In *IJCAI*, pages 557–564, 2001.

[12] Tevfik Bultan. Action language: a specification language for model checking reactive systems. In *International Conference on Software Engineering*, pages 335–344, 2000.

[13] R. Campbell and A. Habermann. The specification of process synchronisation by path expressions. In *Proceedings of an International Symposium on Operating Systems*, pages 89–102. Springer-Verlag, April 1974.

[14] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1):82–126, March 1997.

[15] J. Koehler and R. Treinen. Constraint deduction in an interval-based temporal logic. In M. Fisher and R. Owens, editors, *Proc. of the IJCAI'93 Workshop*, pages 103–117. Springer, 1995. volume 897 of LNAI.

[16] Vijay Saraswat, Radha Jagadeesan, and Vinheet Gupta. Programming in timed concurrent constraint languages. In B. Mayoh, E. Tyugu, and J. Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, pages 361–410, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1994. Springer Verlag.

[17] http://www.cwi.nl/projects/alp/newsletter/nov01/nav/palamidessi/.

[18] Morgens Nielsen, Catuscia Palamidessi, and Frank D. Valencia. On the expressive power of temporal concurrent constraint programming languages. In *PPDP*, October 2002.

[19] Rina Dechter. *Constraint Processing*. Morgan Kauffman, 2003. ISBN 1-55860-890-7.

[20] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995. ISBN 0-13-103805-2.

[21] Kim Marriott and Peter Stuckey. *Programming With Constraints: An Introduction*. The MIT Press, 1998. ISBN 0-262-13341-5.

[22] Vijay Saraswat and Pascal van Hentenryck. *Principles and Practice of Constraint Programming*. The MIT Press, 1995. ISBN 0-262-19361-2.

[23] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004. ISBN 0-262-22069-5.

[24] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

[25] G. Nelson and D.C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[26] R. Shostak. On the SUP-INF method for proving presburger formulas. *Journal of the ACM*, 24(4):529–543, October 1977.

[27] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, 2002. Copenhagen, Denmark.

[28] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[29] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *CACM*, 5(7):394 – 397, July 1962.

[30] J. Silva and K. Sakallah. GRASP – A new search algorithm for satisfiability. In *International Conference on Computer Aided Design*, pages 220–227, 1996. ISBN:0-8186-7597-7.

[31] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *Computer Aided Verification*, pages 17–36, 2002. LNCS 2402.

[32] R. Bryant, S. Lahiri, and S. 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. 14th Intl. Conference*, pages 78–92, 2002.

[33] W. Harvey and P. J. Stuckey. A unit two variable per inequality integer constraint solver for constraint logic programming. In *Proceedings of the Twentieth Australasian Computer Science Conference (ACSC'97)*, pages 102–111, Macquarie University, Sydney, 1997.

[34] Byron Cook, November 2003. Static Driver Verifier, presented at the Driver Developers Conference (DDC), Redmond, WA, 11-14 November. See also http://research.microsoft.com/users/bycook/proj.

[35] L. Michel and P. Van Hentenryck. A modeling layer for constraint programming libraries. Technical Report CS-00-07, Brown University, December 2000.

[36] M. Torrens, R. Weigel, and B. Faltings. Java constraint library: Bringing constraints technology on the internet using the java language. In *In Constraints and Agents: Papers from the 1997 AAAI Workshop*, pages 21–25. AAAI Press, 1997.

[37] N. Barnier and P. Brisset. Facile : A functional constraint library, 2001.

[38] http://ai.uwaterloo.ca/~vanbeek/software/software.html.

[39] http://gnu-prolog.inria.fr/.

[40] http://www.sics.se/isl/sicstus.html.

[41] Tobias Muller. Promoting constraints to first-class status. In *Proc. First International Conference on Computational Logic*, pages 429–447, 2000. LNCS 1861—CL2000.

[42] Pascal Van Hentenryck and Laurent Michel. Control abstractions for local search. In *Principles and Practice of Constraint Programming*, pages 65–80. Springer-Verlag, November 2003. LNCS 2833.

[43] G. Berry and L. Cosserat. The synchronous programming language esterel and its mathematical semantics. In *Seminar on Concurrency, Springer-Verlag LNCS 197*, pages 389–448, 1984.

[44] Daphne Koller. Probabilistic relational models. In Saso Dzeroski and Peter Flach, editors, *Inductive Logic Programming, 9th International Workshop (ILP-99)*, pages 3–13. Springer Verlag, 1999.

[45] Patrice Godefroid, Michael Huth, and Radha Jagadeesan. Abstraction-based model checking using modal transition systems. In *CONCUR*, pages 426+, 2001. Lecture Notes in Computer Science, 2154.

[46] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003. ISBN 0-32122-862-6.

An Operational Semantics for Safety PSL
DRAFT

Koen Claessen[1,2] and Johan Mårtensson[1,3]
{koen,johan}@safelogic.se
[1]Safelogic AB [2]Chalmers University of Technology [3]Gothenburg University

February 20, 2004

Abstract

Extending linear temporal logic by adding regular expressions increases its expressiveness. However, as for example problems in recent versions of Accellera's Property Specification Language as well as in OpenVera's ForSpec and other property languages show, it is a non-trivial task to give a formal denotational semantics with desirable properties to the resulting logic. In this paper, we argue that specifying an *operational semantics* may be helpful in guiding this work, and as a bonus leads to an implementation of the logic for free. We give a concrete operational semantics to the safety property subset of PSL, and show that it is sound and complete with respect to a new denotational semantics proposed in a recent work.

1 Introduction

Accellera and PSL Accellera [1] is an organization set up by major players in the electronic design industry with the objective to promote the use of standards in this industry. In spring 2003, a standard property specification language for hardware designs was agreed upon, called PSL [2]. The standard defines the syntax and semantics of PSL formally. A new version of the language, PSL 1.1 is scheduled to appear in spring 2004.

The logical core of PSL consists of standard Linear Temporal Logic (LTL) constructs augmented with regular expressions. Thus, PSL contains the notion of *formula*, which is an LTL entity that can be satisfied by an infinite sequence of letters, and the notion of *expression*, which is a regular expression that can only be satisfied by a finite sequence of letters. A letter simply defines the values of all variables at one point in time. An expression is also called SERE, which stands for Sequential Extended Regular Expression.

Expressions can be converted into formulas, by using for example the *weak embedding* of an expression r, written $\{r\}$[1]. In both PSL 1.01 and 1.1, a sequence s makes $\{r\}$ true, if there is a finite prefix of s that satisfies r or all finite prefixes of s can be extended to satisfy r.

[1]In fact, this is only valid PSL 1.1 syntax and not valid PSL 1.01 syntax, but this construct is nevertheless expressible in PSL 1.01.

Further, the semantics of PSL has to cope with the fact that properties are supposed to be used both in *static verification* — checking that a property holds solely by analyzing the design — and in *dynamic verification* — checking that a property holds for a concrete and finite trace of the design. To deal with dynamic verification, satisfiability is extended to finite (*truncated*) sequences even for formulas [4, 5].

Anomalies The semantics of the current and forthcoming versions of PSL (1.01 and 1.1) are given by means of denotational semantics. One problem that may arise when using a denotational semantics is that it may be far from obvious what the logical constructs in the language should mean in all cases. Making a seemingly intuitively correct decision can lead to undesirable properties of the resulting logic.

For example in PSL 1.01 the formula $\{[*]; a\}$, which is the weak embedding of an expression that is satisfied by any sequence ending with the atom a, is satisfied by any sequence that makes a always false. However, the formula $\{[*]; F\}$ (where we have simply replaced a by the false constant F) is not satisfied by any sequence. Also $\{[*]; F\}$ is not satisfied even if it is aborted at the first instance. (For a discussion see [4, 3].)

In response to this, Accellera has developed a different semantical paradigm that is proposed for the next iteration, PSL 1.1. In this semantics, the notion of model is changed by introducing a new semantical concept; a special letter $\top$ that can satisfy any one-letter expression, regardless if it is contradictory or not. Unfortunately, PSL 1.1 suffers from a similar anomaly F and $\{a\&\&\{a; a\}\}$ (a so-called *structural contradiction*) are equivalent in an intuitive sense (they cannot be satisfied on actual runs of a system), but are not interchangeable in formulas. (This peculiarity is not unusual; it is for example also present in ForSpec's reset semantics.)

There is work underway within Accellera to deal with this anomaly either by discouraging the use of particular "degenerated formulas", or by extending the model concept further to include models on which structural contradictions are satisfied.

Operational Semantics The company Safelogic develops tools for static and dynamic verification of PSL properties of designs. In order to understand the semantical issues involved in PSL, and to be able to implement our tools, we defined a *structural operational semantics* for a subset of PSL. This subset is precisely the subset of PSL in which safety properties can be expressed.

Our operational semantics is a small-step letter-by-letter semantics with judgments of the form $\phi \xrightarrow{\ell} \psi$. The intention is that in order to check if a sequence s starting with the letter ℓ satisfies ϕ, we simply check that the tail of s (without ℓ) satisfies ψ, and so on. The operational semantics can directly be used for implementing dynamic verification of properties, and also forms the basis of the implementation of our static verification engine.

When specifying a structural operational semantics, there are far less choices to be made than in a denotational semantics, so there is less room for mistakes.

In a recent work a new denotational semantics for LTL with regular expressions on truncated words has been investigated. This semantics could arguably be extended to a full PSL semantics that fixes the anomalies in the semantics of PSL 1.0 and 1.1 [5]. We have shown that our operational semantics is sound

and complete on the weak fragment of PSL with respect to an extension of this denotational semantics to full PSL. Hopefully, the next iteration of PSL will adopt the proposals made in this new semantics!

This Paper The rest of this paper is organized as follows. In Section 2, we specify the safety property subset of PSL we have been working with. In Section 3, we define a structural operational semantics for this language. In Section 4, we present the denotational semantics for our subset of PSL, corresponding to [5]. In Section 5, we show lemmas relating the two semantics, and state soundness and completeness of our operational semantics. Section 6 concludes.

2 Weak Property Language

In this section, we identify a subset of PSL, called Weak Property Language (WPL). This subset can only be used to write safety properties.

As is done in the PSL Language Reference Manual [2], we start by assuming a non-empty set P of *atomic propositions*, and a set of boolean expressions B over P. We assume two designated boolean expressions $\mathbf{tt}$, $\mathbf{ff}$ belonging to B. We define two syntactical subclasses for WPL: expressions and formulas.

Definition 1 (ERE) *If $b \in B$, the language of Extended Regular Expressions (EREs) r has the following grammar:*

$$r ::= \bot \mid \varepsilon \mid b \mid r_1; r_2 \mid r_1 | r_2 \mid r_1 \&\& r_2 \mid r*.$$

The expression $\bot$ denotes the expression with the empty language, ε is the expression that only contains the empty word (see Section 4.4 for an explanation of why those expressions were introduced), $r_1; r_2$ stands for sequential composition between r_1 and r_2, $r_1|r_2$ stands for choice, $r_1\&\&r_2$ stands for intersection, r^* is the Kleene star.

Definition 2 (WPL) *If r, r_1 and r_2 are EREs, and b a boolean expression, the language of WPL formulas ϕ and ψ has the following grammar:*

$$\phi, \psi ::= \{r\} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid X\phi \mid \phi_1 W \phi_2 \mid r \models \phi \mid \phi \,\mathbf{abort}\, b.$$

The formula $\{r\}$ is the weak embedding of the expression r, $\phi_1 \wedge \phi_2$ is formula conjunction, $\phi_1 \vee \phi_2$ is formula disjunction, $X\phi$ is the next operator, $\phi_1 W \phi_2$ is the weak until operator, $r \mid\Rightarrow \phi$ is suffix implication, and $\phi \,\mathbf{abort}\, b$ is the abort operator. When relating this language to full PSL we assume the usual definitions of weak operators in terms of their strong counterparts ($X\phi = \neg X! \neg \phi$ and $\phi W \psi = \neg(\neg\psi U(\neg\phi \wedge \neg\psi))$).

Weak suffix implication $r \mid\Rightarrow \phi$ is satisfied by a word if whenever r accepts a prefix of the word, the formula ϕ holds on the rest of that word. The formula $\phi \,\mathbf{abort}\, b$ is satisfied by a word if ϕ is not made false by that word until b holds.

A formal definition of these constructs is given, by means of an operational semantics in the next section, and by means of a denotational semantics in the section thereafter.

3 A Structural Operational Semantics for WPL

Before we can give the rules of the operational semantics, we need to define a few helper functions that identify syntactic properties of expressions and formulas.

3.1 Preliminaries

First, we define a function em that calculates if a given ERE can accept the empty word.

Definition 3 *We define (inductively) for EREs:*

$$\begin{aligned}
\mathsf{em}(\bot) &= 0 \\
\mathsf{em}(\varepsilon) &= 1 \\
\mathsf{em}(b) &= 0 \\
\mathsf{em}(r_1; r_2) &= \min(\mathsf{em}(r_1), \mathsf{em}(r_2)) \\
\mathsf{em}(r_1 | r_2) &= \max(\mathsf{em}(r_1), \mathsf{em}(r_2)) \\
\mathsf{em}(r_1 \&\& r_2) &= \min(\mathsf{em}(r_1), \mathsf{em}(r_2)) \\
\mathsf{em}(r^*) &= 1
\end{aligned}$$

Second, we define a function ok that conservatively calculates whether a given expression or formula can possibly still be satisfied. Such an expression or formula is said to be OK.

Definition 4 *We define (inductively) for EREs*

$$\begin{aligned}
\mathsf{ok}(\bot) &= 0 \\
\mathsf{ok}(\varepsilon) &= 1 \\
\mathsf{ok}(b) &= 1 \\
\mathsf{ok}(r_1; r_2) &= \mathsf{ok}(r_1) \\
\mathsf{ok}(r_1 | r_2) &= \max(\mathsf{ok}(r_1), \mathsf{ok}(r_2)) \\
\mathsf{ok}(r_1 \&\& r_2) &= \min(\mathsf{ok}(r_1), \mathsf{ok}(r_2)) \\
\mathsf{ok}(r^*) &= 1
\end{aligned}$$

Definition 5 *We define (inductively) for WPLs*

$$\begin{aligned}
\mathsf{ok}(\{r\}) &= \mathsf{ok}(r) \\
\mathsf{ok}(\phi_1 \wedge \phi_2) &= \min(\mathsf{ok}(\phi_1), \mathsf{ok}(\phi_2)) \\
\mathsf{ok}(\phi_1 \vee \phi_2) &= \max(\mathsf{ok}(\phi_1), \mathsf{ok}(\phi_2)) \\
\mathsf{ok}(X\phi) &= 1 \\
\mathsf{ok}(\phi_1 W \phi_2) &= \max(\mathsf{ok}(\phi_1), \mathsf{ok}(\phi_2)) \\
\mathsf{ok}(r \models\!\!> \phi) &= 1 \\
\mathsf{ok}(\phi \,\mathbf{abort}\, b) &= \mathsf{ok}(\phi)
\end{aligned}$$

It is obvious that $\mathsf{em}(\phi)$ and $\mathsf{ok}(\phi)$ is either 0 or 1. We will often write $\mathsf{em}(\phi)$ and $\mathsf{ok}(\phi)$ for $\mathsf{em}(\phi) = 1$ and $\mathsf{ok}(\phi) = 1$ and $\neg\, \mathsf{em}(\phi)$ and $\neg\, \mathsf{ok}(\phi)$ for $\mathsf{em}(\phi) = 0$ and $\mathsf{ok}(\phi) = 0$.

We end with the following observation, which is that any expression or formula accepting the empty string is an OK expression or formula.

Lemma 1 (Empty is OK) *For all EREs r*

$$\mathsf{em}(r) \Rightarrow \mathsf{ok}(r).$$

3.2 The Operational Rules

Let Σ be the set of subsets of P. The elements of Σ can be seen as valuations of the atomic propositions, and are called *letters*. We assume that there is a relation $\Vdash \subseteq \Sigma \times B$ of satisfaction, such that for all letters $\ell \in \Sigma$ $\ell \Vdash \mathbf{tt}$ and $\ell \not\Vdash \mathbf{ff}$.

Our operational semantics has judgments of the form $\phi \xrightarrow{\ell} \psi$, which means that in order to check if a word starting with the letter ℓ satisfies ϕ, one can just as well check that ψ is satisfied by the word without the first letter. The lemma stating this property is referred to as the stepping lemma (Lemma 17).

We start by giving rules for the basic EREs.

$$\text{(Bool)} \qquad b \xrightarrow{\ell} \begin{cases} \varepsilon & \text{if } \ell \Vdash b \\ \bot & \text{otherwise} \end{cases}$$

$$\text{(Bot)} \qquad \bot \xrightarrow{\ell} \bot$$

$$\text{(Empty)} \qquad \varepsilon \xrightarrow{\ell} \bot$$

Here are the rules for sequential composition. There are two cases, one for the case when r_1 can not accept the empty word, and one for the case when it can.

$$\text{(Seq1)} \qquad \frac{r_1 \xrightarrow{\ell} r_1'}{r_1; r_2 \xrightarrow{\ell} r_1'; r_2} \quad \text{not } \mathsf{em}(r_1)$$

$$\text{(Seq2)} \qquad \frac{r_1 \xrightarrow{\ell} r_1' \qquad r_2 \xrightarrow{\ell} r_2'}{r_1; r_2 \xrightarrow{\ell} (r_1'; r_2) | r_2'} \quad \mathsf{em}(r_1)$$

Here are the rules for choice and intersection.

$$\text{(EreOr)} \qquad \frac{r_1 \xrightarrow{\ell} r_1' \qquad r_2 \xrightarrow{\ell} r_2'}{r_1 | r_2 \xrightarrow{\ell} r_1' | r_2'}$$

$$\text{(EreAnd)} \qquad \frac{r_1 \xrightarrow{\ell} r_1' \qquad r_2 \xrightarrow{\ell} r_2'}{r_1 \&\& r_2 \xrightarrow{\ell} r_1' \&\& r_2'}$$

And lastly, this is the rule for Kleene star.

$$\text{(Star)} \qquad \frac{r \xrightarrow{\ell} r'}{r^* \xrightarrow{\ell} r'; r^*}$$

Embedding of expressions simply parses the ℓ through the expression.

$$\text{(ERE)} \qquad \frac{r \xrightarrow{\ell} r'}{\{r\} \xrightarrow{\ell} \{r'\}}$$

Formula disjunction and conjunction are identical to their expression counterparts.

$$\text{(WPLAND)} \qquad \frac{\phi_1 \xrightarrow{\ell} \phi_1' \quad \phi_2 \xrightarrow{\ell} \phi_2'}{\phi_1 \wedge \phi_2 \xrightarrow{\ell} \phi_1' \wedge \phi_2'}$$

$$\text{(WPLOR)} \qquad \frac{\phi_1 \xrightarrow{\ell} \phi_1' \quad \phi_2 \xrightarrow{\ell} \phi_2'}{\phi_1 \vee \phi_2 \xrightarrow{\ell} \phi_1' \vee \phi_2'}$$

The rule for next simply drops the next operator.

$$\text{(NEXT)} \qquad X\phi \xrightarrow{\ell} \phi$$

The rule for weak until is directly derived from the fact that weak until is a solution of the following equation: $\phi_1 W \phi_2 = \phi_2 \vee (\phi_1 \wedge X(\phi_1 W \phi_2))$.

$$\text{(UNTIL)} \qquad \frac{\phi_1 \xrightarrow{\ell} \phi_1' \quad \phi_2 \xrightarrow{\ell} \phi_2'}{\phi_1 W \phi_2 \xrightarrow{\ell} \phi_2' \vee (\phi_1' \wedge (\phi_1 W \phi_2))}$$

For suffix implication, there are two rules: one that triggers the formula ϕ to be true when r accepts the empty word, and one that does not triggers ϕ. One can see these rules as dual to the rules for sequential composition.

$$\text{(WSI1)} \qquad \frac{r \xrightarrow{\ell} r' \quad \phi \xrightarrow{\ell} \phi'}{r \models\!\Rightarrow \phi \xrightarrow{\ell} r' \models\!\Rightarrow \phi} \text{ not } \mathsf{em}(r)$$

$$\text{(WSI2)} \qquad \frac{r \xrightarrow{\ell} r' \quad \phi \xrightarrow{\ell} \phi'}{r \models\!\Rightarrow \phi \xrightarrow{\ell} (r' \models\!\Rightarrow \phi) \wedge \phi'} \ \mathsf{em}(r)$$

An abort checks its formula until the boolean becomes true.

$$\text{(ABORT1)} \qquad \frac{\phi \xrightarrow{\ell} \phi'}{\phi \,\mathbf{abort}\, b \xrightarrow{\ell} \phi' \,\mathbf{abort}\, b} \text{ not } \mathsf{ok}(\phi) \text{ or } \ell \not\Vdash b$$

$$\text{(ABORT2)} \qquad \phi \,\mathbf{abort}\, b \xrightarrow{\ell} \mathbf{tt}^* \quad \mathsf{ok}(\phi) \text{ and } \ell \Vdash b$$

We can see that the operational rules for abort follow the operational intuition behind the operation.

3.3 Iterated Application of Rules

We are interested in the result of applying the rules above iteratively to formulas with respect to words from the the alphabet Σ. A word is a finite or infinite enumeration of letters from Σ. We use ϵ to denote the empty word.

We use juxtaposition to denote concatenation: if w and v are words and w is finite then wv is the concatenation of w and v, i.e. if $w = (\ell_0, \ldots, \ell_n)$ and $v = (\ell'_0, \ldots (, \ell'_n))$ then $wv = (\ell_0, \ldots, \ell_n, \ell'_0, \ldots (, \ell'_n))$. If w is infinite then wv is w. We observe that concatenation is associative, i.e. $w(vu) = (wv)u$ for all w, v and u, and ϵ is the identity, i.e. $\epsilon w = w\epsilon = w$ for all w. We will use ℓ both for denoting the letter ℓ and the word consisting of the single letter ℓ.

Word indexing is defined as follows. If $i < |w|$ then w^i is the $i+1^{st}$ letter of w. $w^{i\cdots}$ is the suffix of w starting at i. If $i \geq |w|$ then $w^{i\cdots} = \epsilon$. If $k \leq j < |w|$, then $w^{k\cdots j}$ means $(w^k, \ldots, w^j)$. If $j < k < |w|$, then $w^{k\cdots j}$ is ϵ.

We use $v \leq w$ and 'v is a *prefix* of w' to say that there is a u such that $vu = w$ and $v < w$ to say that $v \leq w$ and $v \neq w$.

Iteratively applying the operational semantics on a formula ϕ over the letters of a word w is written $\phi\langle w\rangle$:

Definition 6 (After a Word) *For an ERE or WPL p, we define the result $p\langle w\rangle$ of applying the operational rules on p with respect to a finite word w.*

- $p\langle\epsilon\rangle = p$
- $p\langle\ell w\rangle = p'\langle w\rangle$ *where* $p \xrightarrow{\ell} p'$.

We observe the following properties of this relation.

Lemma 2 (Chaining of Eating) *For all EREs and WPLs p, and all finite words w and v, $p\langle w\rangle\langle v\rangle = p\langle wv\rangle$.*

Lemma 3 (After One Step1) *For all EREs and WPLs p, and ℓ and finite words w,*

$$p \xrightarrow{\ell} p\langle\ell\rangle.$$

From Lemmas 2 and 3 we get the following lemma.

Lemma 4 (After One Step2) *For all EREs and WPLs p, and ℓ and finite words w,*

$$p\langle w\rangle \xrightarrow{\ell} p\langle w\ell\rangle.$$

The operational semantics preserves conjunctions and disjunctions.

Lemma 5 (Conservation of Disjuncts) *For all finite words w and WPLs ϕ and ψ*

$$(\phi \vee \psi)\langle w\rangle = \phi\langle w\rangle \vee \psi\langle w\rangle$$

and EREs r_1 and r_2

$$(r_1|r_2)\langle w\rangle = r_1\langle w\rangle|r_2\langle w\rangle.$$

Lemma 6 (Conservation of Conjuncts) *For all finite words w and WPLs ϕ and ψ*

$$(\phi \wedge \psi)\langle w\rangle = \phi\langle w\rangle \wedge \psi\langle w\rangle$$

and EREs r_1 and r_2

$$(r_1\&\&r_2)\langle w\rangle = r_1\langle w\rangle\&\&r_2\langle w\rangle.$$

Finally, we can relate the ok function to this relation.

Lemma 7 (Conservation of Misery) *For each WPL or ERE p, we have*

$$\neg\,\mathsf{ok}(p) \Rightarrow \text{ for all finite } u\ \neg\,\mathsf{ok}(p\langle u\rangle).$$

3.3.1 The Operational Semantics

Now we are ready to define what it means for a formula to be true according to the operational semantics.

Definition 7 (The Operational Semantics) *For all WPLs and ERSs p, and all words w we define*

$$w \vdash p \Leftrightarrow \text{ for all finite } v \text{ such that } v \leq w, \mathsf{ok}(p\langle v \rangle).$$

Intuitively, this means that a word w makes a formula ϕ true if and only if iteratively applying the operational semantics on ϕ using w only produces OK formulas.

We observe the following consequence of Lemma 7.

Lemma 8 *For all WPLs and ERSs p, if w is finite*

$$w \vdash p \Leftrightarrow \mathsf{ok}(p\langle w \rangle).$$

Conjunction and disjunction turn out to be compositional w.r.t. the operational semantics, which is a direct consequence of Lemmas 7, 5 and 6.

Lemma 9 (Operational Disjunction is Compositional)

$$w \vdash r_1 | r_2 \Leftrightarrow w \vdash r_1 \texttt{ or } w \vdash r_2$$

$$w \vdash \phi \vee \psi \Leftrightarrow w \vdash \phi \texttt{ or } w \vdash \psi$$

Lemma 10 (Operational Conjunction is Compositional)

$$w \vdash r_1 \&\& r_2 \Leftrightarrow w \vdash r_1 \texttt{ and } w \vdash r_2$$

$$w \vdash \phi \wedge \psi \Leftrightarrow w \vdash \phi \texttt{ and } w \vdash \psi$$

4 Denotational Semantics

Alternatively, we can define a denotational semantics for WPL. The following definitions are inspired by [5]. In Section 4.4 we describe the relation between this semantics and that of [5].

4.1 Weak and Neutral Words

Let N denote the set of finite and infinite words over Σ, and $N^f \subset N$ the set of finite words over Σ. The elements of N are called *neutral words*. Let $W = \{u^- | u \in N^f\}$. Whenever the notation u^- is used, it is understood that $u \in N^f$. The elements of W are called *weak words*. Note in particular that $\epsilon^- \in W$.

Let $A = N \cup W$, and define concatenation in A as follows. For all $u, v \in N$, uv is equal to the concatenation in N, and if u is finite then $u(v^-) = (uv)^-$. For all $u, v \in A$, if u is infinite or $u \in W$ then $uv = u$. With this definition concatenation in A is associative and ϵ is the unique identity element. Define the length of an element w in N as the number of letters in w if w is finite and ω otherwise, and in A according to $|u^-| = |u|$ for all $u \in N^f$.

Word indexing in A is defined as follows. For $i < |w|$, $(w^-)^i = w^i$. We let $(w^-)^{i\cdots} = (w^{i\cdots})^-$. We also let $(w^-)^{k\cdots j} = w^{k\cdots j}$.

4.2 Tight Satisfaction

We start by giving a definition of *tight satisfaction* $\models$ for EREs on finite words from A.

Definition 8 *Let r, r_1 and r_2 denote EREs, and b a boolean and $w, w_1, \ldots, w_j$ words in A.*

$w \not\models \bot$

$w \models b \Leftrightarrow$ *either* $w = \epsilon^-$ *or* $(|w| = 1$ *and* $w^0 \Vdash b)$

$w \models r_1; r_2 \Leftrightarrow$ *there are* w_1, w_2 *such that* $w = w_1 w_2$ *and* $w_1 \models r_1$ *and* $w_2 \models r_2$

$w \models r_1 | r_2 \Leftrightarrow w \models r_1$ *or* $w \models r_2$

$w \models r_1 \&\& r_2 \Leftrightarrow w \models r_1$ *and* $w \models r_2$

$w \models r^* \Leftrightarrow$ *either* $w = \epsilon$ *or there exists* $w_1, w_2, \ldots, w_j$ *such that* $w = w_1 w_2 \cdots w_j$ *and for all* i *such that* $1 \leq i \leq j, w_i \models r$

$w \models \varepsilon \Leftrightarrow w \models \mathbf{ff}^*$

We note the following lemmas.

Lemma 11 *For all EREs r that do not syntactically contain $\bot$ as a subexpression*

$$\epsilon^- \models r.$$

Lemma 12 *For all EREs r and $w \in A$*

$$w \models r \text{ and } v \leq w \Rightarrow v^- \models r.$$

4.3 Formula Satisfaction

We now define *formula satisfaction* $\vDash$ for WPLs on words from N.

Definition 9 *Let ϕ and ψ denote WPLs, and b a boolean, r an ERE and w, u, v etc. are words in N*

$w \vDash \{r\} \Leftrightarrow$ *for all finite u such that $u \leq w, u^- \models r; \mathbf{tt}^*$*

$w \vDash \phi \wedge \psi \Leftrightarrow w \vDash \phi$ *and* $w \vDash \psi$

$w \vDash \phi \vee \psi \Leftrightarrow w \vDash \phi$ *or* $w \vDash \psi$

$w \vDash X\phi \Leftrightarrow$ *if* $|w| \geq 1$ *then* $w^{1\cdots} \vDash \phi$

$w \vDash \phi W \psi \Leftrightarrow$ *for all k such that $w^{k\cdots} \not\vDash \phi$ there is $j \leq k$ such that $w^{j\cdots} \vDash \psi$*

$w \vDash r \models\!\!> \phi \Leftrightarrow$ *for all u, v such that $uv = w$ if $u \models r$ then $v \vDash \phi$*

$w \vDash \phi \,\mathbf{abort}\, b \Leftrightarrow$ *either $w \vDash \phi$ or there is $k < |w|$ such that $w^k \Vdash b$ and $(w^{0\ldots k-1}) \vDash \phi$*

It follows by structural induction from Lemma 11:

Lemma 13 *For all WPLs ϕ that do not syntactically contain $\bot$ as a subexpression*

$$\epsilon \vDash \phi.$$

We note the following.

Observation 1 *For all w*

$$w \vDash \mathbf{tt}^*; (\mathbf{tt}\&\&(\mathbf{tt}; \mathbf{tt})).$$

Take any finite prefix v of w then $v| \equiv \mathbf{tt}^*$ and $\epsilon^- | \equiv (\mathbf{tt}\&\&(\mathbf{tt}; \mathbf{tt}))$ so $v^- | \equiv \mathbf{tt}^*; (\mathbf{tt}\&\&(\mathbf{tt}; \mathbf{tt}))$.

4.4 Relations to Truncated Ere Semantics

The definition of tight satisfaction above is equivalent to the one given in [5] reduced to the set $N^f \cup W$ except for the following. In the definition of tight satisfaction on weak/strong words in [5] we have

$$w \models' b \Leftrightarrow \text{ either } w = \epsilon^- \text{ or } (w \in N \text{ and } |w| = 1 \text{ and } w^0 \Vdash b)$$

The current formulation

$$w \models b \Leftrightarrow \text{ either } w = \epsilon^- \text{ or } (w \in (N \cup W) \text{ and } |w| = 1 \text{ and } w^0 \Vdash b)$$

results in $w \models r \Leftrightarrow w \models r; \varepsilon$, whereas if $\ell \Vdash b$, $\ell^- \models' b; \varepsilon$ but $\ell^- \not\models' b$. The change was necessary to get Lemma 16 and it can be done without losing desirable properties like Lemma 12 that are needed to get the results of [5]. We also introduced the ERE symbols $\perp$ and ε that are not present in [5]. It was necessary to differentiate falsity that is already visited ($\perp$ which should be false on ϵ^-) from falsity that is not already visited (**ff** which should be true on ϵ^-) in the operational rules to get Lemma 16. It was also convenient for defining the operational rules in a succinct way to introduce a symbol ε that is only tightly satisfied by empty words.

In [5] both tight and formula satisfaction is defined w.r.t. to the set $A = N \cup W \cup S$ of neutral, weak and strong words. The strong words are needed for defining the semantics of negation on weak words. Our weak language does not contain negation because the negation of weak formulas are strong formulas. We have defined tight satisfaction on the set $N^f \cup W$ and formula satisfaction on the set N. Formula satisfaction on weak words is used in [5] in the case for the **abort** (trunc_w) operator. We don't need that because a weak formula is satisfied on a weak word if and only if it is satisfied on its neutral counterpart.

In order to better see the relationship between the definition above and that of [5][2] we introduce for $k > 0$ the operator X^k as syntactic sugar for iterated applications of the X operator: We thus let $X^1\phi = \phi$ and $X^{k+1}\phi = X(X^k\phi)$.

We observe the following two facts.

Lemma 14

$$w \vDash X^k\phi \Leftrightarrow \textit{ if } |w| \geq k \textit{ then } w^{k\cdots} \vDash \phi.$$

Lemma 15

$$w \vDash \phi W \psi \Leftrightarrow \textit{ for all } k \textit{ such that } w \nvDash X^k\phi \textit{ there is } j \leq k \textit{ such that } w \vDash X^j\psi.$$

[2]The relations to the semantics of [5] will be made more explicit in the final version of this paper

5 Relations Between the Semantics

In this section we show that the operational semantics and denotational semantics are tightly coupled.

5.1 The Stepping Lemmas

We can show the following two basic lemmas, which confirms our intuition about the operational judgments $r \xrightarrow{\ell} r'$ and $\phi \xrightarrow{\ell} \phi'$.

Lemma 16 (ERE Stepping) *If $r \xrightarrow{\ell} r'$ then for all $w \in A$*

$$\ell w \models\!\!\models r \Leftrightarrow w \models\!\!\models r'.$$

Using Lemma 16 we can prove the following.

Lemma 17 (WPL Stepping) *If $\phi \xrightarrow{\ell} \phi'$ then for all $w \in A$*

$$\ell w \vDash \phi \Leftrightarrow w \vDash \phi'.$$

5.2 Completeness

In order to show completeness of the operational semantics with respect to the denotational semantics, we first observe the following.

Lemma 18 (Tight True is Ok) *For $w \in A$*

$$w \models\!\!\models r \Rightarrow \mathsf{ok}(r).$$

The following lemma follows.

Lemma 19 (True is Ok) *For $w \in A$*

$$w \vDash \phi \Rightarrow \mathsf{ok}(\phi).$$

We use Lemma 17 and 4 to show the following.

Lemma 20 (True Stays True) *For $w \in A$*

$$w \vDash \phi \Leftrightarrow \text{ for every finite } v \text{ such that } vu = w, u \vDash p\langle v \rangle.$$

Finally, we use Lemmas 20 and 19 to show completeness.

Theorem 1 (Completeness) *For $w \in N$*

$$w \vDash \phi \Rightarrow w \vdash \phi.$$

5.3 Soundness

In order to show soundness of the operational semantics with respect to the denotational semantics, we use Lemmas 1, 5 and 6 to show the following.

Lemma 21 (Empty is Tight) *For all finite* $v \in N$

$$\mathsf{em}(r\langle v\rangle) \Leftrightarrow v \models\!\!\!\models r.$$

Then, we use Lemma 21 and 5 and 7 to show the following.

Lemma 22 (Seq is Sound)

$w \vdash r_1; r_2 \Rightarrow$ *either* $w \vdash r_1$ *or there are* v, u *such that* $vu = w$ *and* $v \models\!\!\!\models r_1$ *and* $u \vdash r_2$.

We use Lemma 22, 16, 11, 9 and 10 to show the following.

Lemma 23 (Tight Soundness) *For all finite* $w \in N$

$$w \vdash r \Rightarrow w^{-} \models\!\!\!\models r.$$

Finally, we use Lemma 23, 17, 9 and 10 to show soundness.

Theorem 2 (Soundness) *For* $w \in N$

$$w \vdash \phi \Rightarrow w \vDash \phi.$$

6 Conclusions and Future Work

We specified our operational semantics independently of the work described in [5]. Nevertheless, we were able to show soundness and completeness of our operational semantics with respect to the one obvious extension of this work to full WPL. The fact that these different approaches coincide is encouraging and indicates that this current semantical definition is the right way to go.

Defining an operational semantics, can help in guiding the work of defining a denotational one, but it also gives a direct way of implementing dynamic property checking, and a basis of an algorithm for static property checking.

Currently, we are working on extending our operational semantics to also deal with non-safety properties. The next step would then be to relate that semantics to the strong satisfiability described in [5].

References

[1] Accellera. www.accellera.org.

[2] Accellera, 1370 Trancas Street, #163, Napa, CA 94558. *Property Specification Language, Reference Manual*, apr 2003. www.accellera.org/pslv101.pdf.

[3] R. Armoni, D. Bustan, O. Kupferman, and M. Vardi. Resets vs. aborts in linear temporal logic. In *TACAS03*, 2003.

[4] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. van Campenhout. Reasoning with temporal logic on truncated paths. In *15th International Conference on Computer Aided Verification (CAV)*, pages 27–39. LNCS 2725, July 2003.

[5] C. Eisner, D. Fisman, J. Havlicek, and J. Mårtensson. Truncating regular expressions. Submitted to CAV04.

Participants' proceedings, DCC 2004, Barcelona, 27-28 March, 2004.

PSL semantics in higher order logic

Mike Gordon, University of Cambridge Computer Laboratory, 23 February 2004

1. Introduction

In a paper, published in the journal *Formal Aspects of Computing* (FAC) [Gor03][1], we described a deep semantic embedding of Version 1.01 of the Accellera Property Specification Language (PSL) in higher order logic. The main goal of that paper was to demonstrate that mechanised theorem proving can be a useful aid to the validation of the semantics of an industrial design language.

In another paper, presented at CHARME 2003 [GHS03], we showed how mechanised deduction could be applied to a formal encoding of the PSL semantics in higher order logic to generate correct-by-construction tools (a property evaluator, a simulation monitor generator and a model checker). The point of that paper was to show that a formal semantics was not just documentation, but could be executed by special purpose theorem proving scripts.

This document gives more detail than the published papers on how the semantics is represented in the HOL system. It also reflects the (not yet released) Version 1.1 semantics. Some material has been taken from the FAC paper, but the details are updated to correspond to the latest version of PSL.

2. Review of higher order logic, the HOL system and semantic embedding

Higher order logic is an extension of first-order predicate calculus that allows quantification over functions and relations. It is a natural notation for formalising informal set theoretic specifications (indeed, it is usually more natural than formal first-order set theories, like ZF). We hope that the formal logic notation in what follows is sufficiently close to standard informal mathematics that it needs no systematic explanation. In this section we briefly outline some features of the version of higher order logic implemented in the HOL4 system. We refer to this logic as "the HOL logic" or just "HOL".

The HOL logic is built out of *terms* which are of four types: constants, variables, combinations (or function applications) $t_1\ t_2$ and λ-abstractions $\lambda x.\ t$.

The particular set of constants that are available depends on the theory one is working in. The kernel of the HOL logic contains constants T and F representing truth and falsity, respectively. In the HOL system, new constants can be defined in terms of existing constants using definitional mechanisms that guarantee no new inconsistencies are introduced. Defined constants include numerals (e.g. $0, 1, 2$), strings (e.g. `"a"`, `"b"`, `"ab"`) and logical operators (e.g. $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$). The details of HOL's theory of definition are available elsewhere [GM93].

The simple kernel of four kinds of terms can be extended using syntactic sugar to include all the normal notations of predicate calculus. The extension process consists of defining new constants and then adding syntactic sugar to make terms containing these constants look familiar. For example, constants $\forall$, $\exists$ and `Pair` can be defined and then $\forall x.\ \exists y.\ P(x, y)$ is syntactic sugar for $\forall(\lambda x.\ \exists(\lambda y.\ P\ (\texttt{Pair}\ x\ y)))$, (here the function application `Pair` $x\ y$ means $((\texttt{Pair}\ x)\ y)$, so `Pair` is 'curried'). If P is a function that returns a truth-value (i.e. a predicate), then P can be thought of as a set, and we write $x \in P$ to mean $P(x)$ is true. The term $\lambda x.\ \cdots x \cdots$ corresponds to the set abstraction $\{x \mid \cdots x \cdots\}$ and we will write $\forall x \in P.\ Q(x)$ and $\exists x \in P.\ Q(x)$ to mean $\forall x.\ P(x) \Rightarrow Q(x)$ and $\exists x.\ P(x) \wedge Q(x)$, respectively.

Address: Mike Gordon, University of Cambridge Computer Laboratory, William Gates Building, JJ Thomson Avenue, Cambridge CB3 0FD, U.K. e-mail: mjcg@cl.cam.ac.uk

[1] Draft online at: `http://www.cl.cam.ac.uk/~mjcg/Sugar/facpaper/`.

Higher order logic is typed to avoid inconsistencies.[2] Types are syntactic constructs that denote sets of values. For example, types *bool* and *num* are atomic types in HOL and denote the sets of booleans and natural numbers, respectively. Complex types can be built using type constructors. For example, if ty_1 and ty_2 are types, then $ty_1 \rightarrow ty_2$ denotes the set of functions with domain ty_1 and range ty_2, and $ty_1 \times ty_2$ denotes the Cartesian product of the sets denoted by ty_1 and ty_2. Type constructors are traditionally applied to their arguments using a postfix notation like $(ty_1, \ldots, ty_n)$*constructor*. The types $ty_1 \rightarrow ty_2$ and $ty_1 \times ty_2$ are just special notations for (ty_1, ty_2)*fun* and (ty_1, ty_2)*prod*, respectively.

If the types for all the variables and constants in a term t are given, then a type-checking algorithm can determine whether t is well-typed – i.e. every function is applied to an argument of the correct type – and compute a type for t. For example, $\neg 3$ is not well-typed (assuming $\neg$ has type $bool \rightarrow bool$ and 3 has type *num*) and would be rejected by type-checking, however, $\neg$`T` is well-typed (assuming `T` has type *bool*) and would be accepted and given type *bool*. Only the well-typed terms are considered meaningful and we write $t : ty$ if term t is well-typed and has type ty. Well-typed terms of type *bool* are the formulas of the HOL logic, thus formulas are a subset of terms: $\forall x.\ \exists y.\ x + 1 < y$ is a term that is a formula, but $x + 1$ is a term (of type *num*) that is not a formula. The HOL logic kernel only has two types and one type constructor: type *bool* of booleans, an infinite type *ind* of 'individuals' and the function type constructor $\rightarrow$. Other types and type constructors can be defined in terms of these [GM93]. For example, the type *num* of numbers is defined as a subset of the primitive type *ind*, and the Cartesian product constructor $\times$ can be defined in terms of $\rightarrow$. Families of terms can be created by using type variables. For example, if variable x is assigned the type α, where α is a type variable, then $\lambda x.\ x$ has type $\alpha \rightarrow \alpha$ and is a family of identity functions with an instance $\lambda x : ty.\ x$ for each type ty.

2.1. HOL system notation

Input to the HOL system uses ASCII characters. The table below shows some common idioms, including those that are used in this paper.

Standard notation	HOL notation	Description
true	`T`	truth
false	`F`	falsity
$\neg t$	`~`t	negation
$t_1 \wedge t_2$	t_1 `/\` t_2	conjunction
$t_1 \vee t_2$	t_1 `\/` t_2	disjunction
$t_1 \Rightarrow t_2$	t_1 `==>` t_2	implication
$\forall x.P(x)$	`!`x`.`$P(x)$	universal quantification
$\exists x.P(x)$	`?`x`.`$P(x)$	existential quantification
$p \in s$	p `IN` s	set membership
`[0..`n`)`	`LESS` n	set of natural numbers less than n
$\forall x \in s.\ P(x)$	`!`x`::s.` $P(x)$	universal quantification restricted to s
$\exists x \in s.\ P(x)$	`?`x`::s.` $P(x)$	existential quantification restricted to s
$\forall x \in$ `[0..`n`).` $P(x)$	`!`x`::LESS` n`.` $P(x)$	universal quantification restricted to numbers less than n
$\exists x \in$ `[0..`n`).` $P(x)$	`?`x`::LESS` n`.` $P(x)$	existential quantification restricted to numbers less than n
ϵ	`[]`	empty list
x	`[`x`]`	list with one element (singleton)
$l_1 l_2$	l_1 `<>` l_2	list concatenation (append)
$ty_1 \times ty_2$	ty_1 `#` ty_2	Cartesian product of types ty_1 and ty_2
$ty_1 \rightarrow ty_2$	ty_1 `-->` ty_2	type of functions from ty_1 to ty_2

To enable an easy comparison with the informal presentation in the PSL Language Reference Manual (LRM), we include snippets from the LRM in framed boxes[3]

[2] Russell's paradox can be formulated as: $(\lambda x.\ \neg(x\ x))\ (\lambda x.\ \neg(x\ x)) = \neg((\lambda x.\ \neg(x\ x))\ (\lambda x.\ \neg(x\ x)))$.

[3] Thanks to Dana Fisman for supplying LaTeX source of the draft LRM. Not that as we are using a different style file for typesetting, the appearance of the material in the boxes may be formatted here differently from how the text will appear in the forthcoming LRM.

2.2. Representing letters and words in HOL

In LRM Version 1.1 (Section B.2.1) we find:

> The semantics of FL is defined with respect to finite and infinite words over $\Sigma = 2^P \cup \{\top, \bot\}$.

Members of Σ are called letters and to represent them in HOL we define a type `(’a)letter`, where the parametrisation on a type variable `’a` is so that different sets P of atomic propositions can be ‘plugged-in’ by instantiating `’a` to a type representing P.

A data-type definition has the form `Hol_datatype ‘<`*description of type*`>‘`. The following input to HOL defines a new type `(’a)letter` together with three constants (which are separated by “`|`”).

```
Hol_datatype ‘letter = TOP | BOTTOM | STATE of (’a -> bool)‘
```

Note that the syntax used for declaring data-types in the HOL system logic requires the type name without any parameters on the left hand side (i.e. `letter` rather than `(’a)letter`). The presence of the single free type variable `’a` in the right hand side causes a unary type operator to be defined.

The constants `TOP` and `BOTTOM` are distinct values of type `(’a)letter`. The constant `STATE` is a function taking an argument of the type shown after the “`of`” and returning a result of type `(’a)letter`. Thus the effect of executing the data-type definition is to define a new type `(’a)letter` together with the following constants.

```
TOP     : (’a)letter
BOTTOM  : (’a)letter
STATE   : (’a → bool) → (’a)letter
```

The argument to `STATE` is the characteristic function of a set of atomic propositions. When HOL performs such a definition it automatically proves a standard set of useful theorems about the type and the constants defined on it (e.g. `~(TOP = BOTTOM)`, which represents $\neg(\top = \bot)$).

The PSL LRM continues:

> We denote a letter from Σ by ℓ and an empty, finite, or infinite word from Σ by u, v, or w (possibly with subscripts).

Finite paths can be represented by a built-in type `list` of finite lists. Infinite paths can be represented as functions from natural numbers (type `num`). Thus to represent paths in HOL we define a disjoint union type:

```
Hol_datatype
 ‘path = FINITE of (’s list) | INFINITE of (num -> ’s)‘
```

This defines a unary type operator `path`. A type `(`*ty*`)path` represents paths whose elements are of type *ty*.

Next the PSL LRM says:

> We denote the length of word v as $|v|$. An empty word $v = \epsilon$ has length 0, a finite word $v = (\ell_0\ell_1\ell_2\cdots\ell_n)$ has length $n + 1$, and an infinite word has length ∞.

The length of a path is thus either a natural number or is ∞. To model this we define a type `xnum` of extended natural numbers. Comments in HOL are enclosed between `(*` and `*)`.

```
Hol_datatype
 ‘xnum = INFINITY                          (* length of an infinite path  *)
       | XNUM of num‘                      (* length of a finite path     *)
```

This defines the type `xnum` together with the following constants.

```
INFINITY : xnum
XNUM     : num → xnum
```

The length of a path can now be defined in the HOL logic by defining a constant `LENGTH : ('a)path → xnum`. The function `list$LENGTH`, which occurs below, is the pre-existing length function on finite lists.

```
Define `(LENGTH(FINITE l)   = XNUM(list$LENGTH l))
        /\
        (LENGTH(INFINITE p) = INFINITY)`
```

This definition overloads the name `LENGTH` so it now can be applied both to lists and to paths.

Continuing with B2.1 of the PSL LRM:

> We use i, j, and k to denote non-negative integers. We denote the i^{th} letter of v by v^{i-1} (since counting of letters starts at zero). We denote by $v^{i\cdot\cdot}$ the suffix of v starting at v^i. That is, for every $i < |v|$, $v^{i\cdot\cdot} = v^i v^{i+1} \cdots v^n$ or $v^{i\cdot\cdot} = v^i v^{i+1} \cdots$. We denote by $v^{i..j}$ the finite sequence of letters starting from v^i and ending in v^j. That is, for $j \geq i$, $v^{i..j} = v^i v^{i+1} \cdots v^j$ and for $j < i$, $v^{i..j} = \epsilon$. We use ℓ^ω to denote an infinite-length word, each letter of which is ℓ.
> We use $\overline{v}$ to denote the word obtained by replacing every $\top$ with a $\bot$ and vice versa. We call $\overline{v}$ the *complement* of v.

These operations are straightforward to define by 'functional programming' in the HOL logic. We do not give the definitions here, but show in the table below the PSL notation and corresponding HOL representation.

PSL Notation	HOL representation	Description
∞	`INFINITY`	infinity
ϵ	`[]`	empty path
$\top^\omega$	`TOP_OMEGA`	infinite repetition of $\top$
$\|v\|$	`LENGTH` v	length of a path
v^i	`ELEM` v i	i+t^{th} letter of v
$v^{i\cdot\cdot}$	`RESTN` v i	suffix of v starting at v^i
$v^{i..j}$	`SEL` v (i,j)	sequence starting at v^i and ending at v^j
$\overline{v}$	`COMPLEMENT` v	complement of v (swap $\top$s and $\bot$s)

3. Representing syntax in higher order logic

PSL has four classes of constructs: boolean expressions, Sequential Extended Regular Expressions (SEREs), Foundation Language (FL) formulas and Optional Branching Extension (OBE) formulas. The OBE is ignored here, though for PSL Version 1.01 its semantics in HOL appears in the FAC paper.

Although the syntax of boolean expressions is not explicitly defined, it says in Section B.1 of the LRM:

> The logic Accellera PSL is defined with respect to a non-empty set of atomic propositions P and a given set of boolean expressions B over P. We assume two designated boolean expression *true* and *false* belong to B.

In addition, in LRM B.2.1 the semantics of boolean expressions $\neg b$ and $b_1 \wedge b_2$ are defined, so we include these as primitives too.

Abstract syntax is represented in HOL by defining a data-type whose operations are the constructors.

For boolean expressions, a data-type `bexp` is defined. Since atomic propositions are boolean expressions, we parameterise the type of boolean expressions on a type variable `'a` that can be subsequently instantiated to a particular type representing the set P of atomic propositions. If `aprop` is such a type, then the type of terms representing boolean expressions is `(aprop)bexp`. Thus `bexp` is a unary type constructor.

When a constructors is to take n arguments, where $n > 1$, one writes "`of` ty_1 `#` $\cdots$ `#` ty_n" after a constructor name in the data-type declaration, where ty_1, ..., ty_n are the types of the arguments.

The input to the HOL system to define `bexp` is:

```
Hol_datatype
 `bexp = B_PROP   of 'a                          (* atomic proposition       *)
       | B_TRUE                                  (* true                     *)
       | B_FALSE                                 (* false                    *)
       | B_NOT    of bexp                        (* negation                 *)
       | B_AND    of bexp # bexp`                (* conjunction              *)
```

This defines a new unary type constructor *bexp* and constants:

```
B_PROP  : 'a → ('a)bexp
B_TRUE  : ('a)bexp
B_FALSE : ('a)bexp
B_NOT   : ('a)bexp → ('a)bexp
B_AND   : ('a)bexp × ('a)bexp → ('a)bexp
```

The prefix `B_` indicates a boolean expression constructor.

If atomic propositions are taken to be strings, then the boolean expression $x \wedge \neg y$ would be represented by the term `B_AND(B_PROP "x", B_NOT(B_PROP "y"))` which has the type `(string)bexp`.

The syntax of SEREs is described in the LRM by:

Definition 1 (Sequential Extended Regular Expressions (SEREs)).

- Every boolean expression $b \in B$ is a SERE.
- If r, r_1, and r_2 are SEREs, and c is a boolean expression, then the following are SEREs:
  - $\{r\}$
  - $r_1 ; r_2$
  - $r_1 : r_2$
  - $r_1 \mid r_2$
  - $r_1 \,\&\&\, r_2$
  - $[*0]$
  - $r[*]$
  - $r@c$

This is represented in HOL by defining a data-type `sere` by (the prefix `S_` indicates a SERE constructor):

```
Hol_datatype
 `sere = S_BOOL        of 'a bexp              (* boolean expression       *)
       | S_CAT         of sere # sere          (* r1 ;  r2                 *)
       | S_FUSION      of sere # sere          (* r1 :  r2                 *)
       | S_OR          of sere # sere          (* r1 |  r2                 *)
       | S_AND         of sere # sere          (* r1 && r2                 *)
       | S_EMPTY                               (* [*0]                     *)
       | S_REPEAT      of sere                 (* r[*]                     *)
       | S_CLOCK       of sere # 'a bexp`      (* r@c                      *)
```

This defines a unary type operator `sere` (the need for parametrisation is inferred from the free type variable `'a` in the right hand side of the definition).

The syntax of FL formulas is defined in the LRM by (the prefix `F_` indicates an FL formula constructor):

Definition 2 (Formulas of the Foundation Language (FL formulas)).

- If b is a boolean expression then both b and $b!$ are FL formulas.
- If φ and ψ are FL formulas, r, r_1, r_2 are SEREs, and b a boolean expression, then the following are FL formulas:
  - (φ)
  - $\neg\varphi$
  - $\varphi \wedge \psi$
  - $r!$
  - r
  - $X!\ \varphi$
  - $[\varphi\ U\ \psi]$
  - φ abort b
  - $r \mapsto \varphi$
  - $\varphi@b$

This is represented in HOL by defining a data-type `fl` by:

```
Hol_datatype
 `fl = F_STRONG_BOOL  of 'a bexp              (* b!                        *)
     | F_WEAK_BOOL    of 'a bexp              (* b                         *)
     | F_NOT          of fl                   (* not f                     *)
     | F_AND          of fl # fl              (* f1 and f2                 *)
     | F_STRONG_SERE  of 'a sere              (* r!                        *)
     | F_WEAK_SERE    of 'a sere              (* r                         *)
     | F_NEXT         of fl                   (* X! f                      *)
     | F_UNTIL        of fl # fl              (* [f1 U f2]                 *)
     | F_ABORT        of fl # 'a bexp         (* f abort b                 *)
     | F_CLOCK        of fl # 'a bexp         (* f@b                       *)
     | F_SUFFIX_IMP   of 'a sere # fl`        (* r  |-> f                  *)
```

This defines a unary type operator `fl`.

4. Formal semantics in higher order logic

In this section we give the semantics that is expected to be released in the forthcoming LRM for Accellera PSL Version 1.1. We then show its representation in HOL, both pretty printed and in raw ASCII form.

4.1. Boolean expressions in PSL

The semantics of boolean expressions is described in the LRM as follows:

> The semantics of boolean expression is assumed to be given as a relation $\models \subseteq \Sigma \times B$ relating letters in Σ with boolean expressions in B. If $(\ell, b) \in \models$ we say that the letter ℓ *satisfies* the boolean expression b and denote it $\ell \models b$. We assume the two special letters $\top$ and $\bot$ behave as follows: for every boolean expression b, $\top \models b$ and $\bot \not\models b$. We assume that otherwise the boolean relation $\models$ behaves in the usual manner. In particular, that for every letter $\ell \in 2^P$, atomic proposition $p \in P$ and boolean expressions $b, b_1, b_2 \in B$ (i) $\ell \models p$ iff $p \in \ell$, (ii) $\ell \models \neg b$ iff $\ell \not\models b$, and (iii) $\ell \models \mathit{true}$ and $\ell \not\models \mathit{false}$. Finally, we assume that for every letter $\ell \in \Sigma$, $\ell \models b_1 \wedge b_2$ iff $\ell \models b_1$ and $\ell \models b_2$.

The semantics of boolean expressions is represented in HOL by defining a new constant corresponding to a semantic function `B_SEM` : `('a→bool)→('a)bexp→bool` such that `B_SEM` l b is true iff b is true with respect to letter l. The actual input to HOL to define `S_SEM` is:

```
Define
 `(B_SEM TOP b = T)
  /\
  (B_SEM BOTTOM b = F)
  /\
  (B_SEM (STATE s) (B_PROP p) = p IN s)
  /\
  (B_SEM (STATE s) B_TRUE = T)
  /\
  (B_SEM (STATE s) B_FALSE = F)
  /\
  (B_SEM (STATE s) (B_NOT b) = ~(B_SEM (STATE s) b))
  /\
  (B_SEM (STATE s) (B_AND(b1,b2)) = B_SEM (STATE s) b1 /\ B_SEM (STATE s) b2)`
```

If `B_SEM l b` is pretty printed as $l \models b$, then the semantics above pretty prints as:

$$
\begin{array}{l}
(\top \Vdash b \;=\; \mathrm{T}) \\
\wedge \\
(\bot \Vdash b \;=\; \mathrm{F}) \\
\wedge \\
(s \Vdash p \;=\; p \in s) \\
\wedge \\
(s \Vdash \mathrm{T} \;=\; \mathrm{T}) \\
\wedge \\
(s \Vdash \mathrm{F} \;=\; \mathrm{F}) \\
\wedge \\
(s \Vdash \neg b \;=\; \neg(s \Vdash b)) \\
\wedge \\
(s \Vdash b_1 \wedge b_2 \;=\; s \Vdash b_1 \wedge s \Vdash b_2)
\end{array}
$$

Pretty-printing introduces potentially confusing overloading: the occurrence of $\neg$ in $\neg b$ is part of the boolean expression syntax of PSL, but the occurrence in $\neg(l \models b)$ is negation in higher order logic. Similarly $\wedge$ is overloaded: the occurrence in $b_1 \wedge b_2$ is part of the boolean expression syntax, but the other occurrences are conjunction in higher order logic.

4.2. Extended Regular Expressions (SEREs)

The unclocked semantics (B.2.1.1.1 of the LRM) is shown in the next box:

Unclocked SEREs are defined over finite words from the alphabet Σ. The notation $v \models\!\models r$, where r is a SERE and v a finite word means that v *models tightly* r. The semantics of unclocked SEREs are defined as follows, where b denotes a boolean expression, and r, r_1, and r_2 denote unclocked SEREs.

- $v \models\!\models \{r\} \iff v \models\!\models r$
- $v \models\!\models b \iff |v| = 1$ and $v^0 \Vdash b$
- $v \models\!\models r_1 \;;\; r_2 \iff \exists v_1, v_2$ s.t. $v = v_1 v_2$, $v_1 \models\!\models r_1$, and $v_2 \models\!\models r_2$
- $v \models\!\models r_1 : r_2 \iff \exists v_1, v_2$, and ℓ s.t. $v = v_1 \ell v_2$, $v_1 \ell \models\!\models r_1$, and $\ell v_2 \models\!\models r_2$
- $v \models\!\models r_1 \mid r_2 \iff v \models\!\models r_1$ or $v \models\!\models r_2$
- $v \models\!\models r_1 \;\&\&\; r_2 \iff v \models\!\models r_1$ and $v \models\!\models r_2$
- $v \models\!\models [*0] \iff v = \epsilon$
- $v \models\!\models r[*] \iff$ either $v \models\!\models [*0]$ or $\exists v_1, v_2$ s.t. $v_1 \neq \epsilon$, $v = v_1 v_2$, $v_1 \models\!\models r$ and $v_2 \models\!\models r[*]$

The pretty-printed HOL representation of this is:

$$
\begin{array}{l}
(v \models\!\models b \;=\; (|v| = 1) \wedge v^0 \Vdash b) \\
\wedge \\
(v \models\!\models r_1; r_2 \;=\; \exists v_1 v_2.\ (v = v_1 v_2) \wedge v_1 \models\!\models r_1 \wedge v_2 \models\!\models r_2) \\
\wedge \\
(v \models\!\models r_1 : r_2 \;=\; \exists v_1 v_2 l.\ (v = v_1[l]v_2) \wedge v_1[l] \models\!\models r_1 \wedge [l]v_2 \models\!\models r_2) \\
\wedge \\
(v \models\!\models r_1 \mid r_2 \;=\; v \models\!\models r_1 \vee v \models\!\models r_2) \\
\wedge \\
(v \models\!\models r_1 \&\& r_2 = v \models\!\models r_1 \wedge v \models\!\models r_2) \\
\wedge \\
(v \models\!\models [*0] \;=\; (v = \epsilon)) \\
\wedge \\
(v \models\!\models r[*] \;=\; v \models\!\models [*0] \vee \exists v_1 v_2.\ \neg(v = \epsilon) \wedge (v = v_1 v_2) \wedge v_1 \models\!\models r \wedge v_2 \models\!\models r[*])
\end{array}
$$

The raw HOL is (we omit the `Define` and enclosing quotes):

```
(US_SEM v (S_BOOL b) = (LENGTH v = 1) /\ B_SEM (ELEM v 0) b)
/\
(US_SEM v (S_CAT(r1,r2)) = ?v1 v2. (v = v1 <> v2) /\ US_SEM v1 r1 /\ US_SEM v2 r2)
/\
(US_SEM v (S_FUSION(r1,r2)) =
  ?v1 v2 l. (v = v1 <> [l] <> v2) /\ US_SEM (v1<>[l]) r1 /\ US_SEM ([l]<>v2) r2)
/\
(US_SEM v (S_OR(r1,r2)) = US_SEM v r1 \/ US_SEM v r2)
/\
(US_SEM v (S_AND(r1,r2)) = US_SEM v r1 /\ US_SEM v r2)
/\
(US_SEM v S_EMPTY = (v = []))
/\
(US_SEM v (S_REPEAT r) =
  US_SEM v S_EMPTY \/
   ?v1 v2. ~(v=[]) /\ (v = v1 <> v2) /\ US_SEM v1 r /\ US_SEM v2 (S_REPEAT r))
```

The clocked semantics (B.2.1.2.1 of the LRM) is more complex.

> We say that finite word v *is a clock tick of* c iff $|v| > 0$ and $v^{|v|-1} \Vdash c$ and for every natural number $i < |v| - 1$, $v^i \Vdash \neg c$.

This is formalised by defining a constant: $\mathsf{ClockTick}(v, c) = |v| > 0 \wedge v^{|v|-1} \Vdash c \wedge \forall i \in [0..|v|-1).\ v^i \Vdash \neg c$.

> Clocked SEREs are defined over finite words from the alphabet Σ and a boolean expression that serves as the clock context. The notation $v \stackrel{c}{\Vdash} r$, where r is a SERE and c is a boolean expression, means that v *models tightly* r *in context of clock* c. The semantics of clocked SEREs are defined as follows, where b, c, and c_1 denote boolean expressions, r, r_1, and r_2 denote clocked SEREs.
>
> - $v \stackrel{c}{\Vdash} \{r\} \iff v \stackrel{c}{\Vdash} r$
> - $v \stackrel{c}{\Vdash} b \iff v$ is a clock tick of c and $v^{|v|-1} \Vdash b$
> - $v \stackrel{c}{\Vdash} r_1\ ;\ r_2 \iff \exists v_1, v_2$ s.t. $v = v_1 v_2$, $v_1 \stackrel{c}{\Vdash} r_1$, and $v_2 \stackrel{c}{\Vdash} r_2$
> - $v \stackrel{c}{\Vdash} r_1 : r_2 \iff \exists v_1, v_2$, and ℓ s.t. $v = v_1 \ell v_2$, $v_1 \ell \stackrel{c}{\Vdash} r_1$, and $\ell v_2 \stackrel{c}{\Vdash} r_2$
> - $v \stackrel{c}{\Vdash} r_1 \mid r_2 \iff v \stackrel{c}{\Vdash} r_1$ or $v \stackrel{c}{\Vdash} r_2$
> - $v \stackrel{c}{\Vdash} r_1 \,\&\&\, r_2 \iff v \stackrel{c}{\Vdash} r_1$ and $v \stackrel{c}{\Vdash} r_2$
> - $v \stackrel{c}{\Vdash} [*0] \iff v = \epsilon$
> - $v \stackrel{c}{\Vdash} r[*] \iff$ either $v \stackrel{c}{\Vdash} [*0]$ or $\exists v_1, v_2$ s.t. $v_1 \neq \epsilon$, $v = v_1 v_2$, $v_1 \stackrel{c}{\Vdash} r$ and $v_2 \stackrel{c}{\Vdash} r[*]$
> - $v \stackrel{c}{\Vdash} r @ c_1 \iff v \stackrel{c_1}{\Vdash} r$

The HOL representation of this semantics of SEREs is defined by a semantic function `S_SEM` such that `S_SEM` w c r is true iff word w is in the language recognised by the extended regular expression r when the clock context (i.e. current clock) is c. The HOL term `S_SEM` w c r is pretty-printed as $w \stackrel{c}{\Vdash} r$.

$$
\begin{array}{l}
(v \stackrel{c}{\Vdash} b \ = \ \mathsf{ClockTick}(v, c) \wedge v^{|v|-1} \Vdash b) \\
\wedge \\
(v \stackrel{c}{\Vdash} r_1; r_2 \ = \ \exists v_1 v_2.\ (v = v_1 v_2) \wedge v_1 \stackrel{c}{\Vdash} r_1 \wedge v_2 \stackrel{c}{\Vdash} r_2) \\
\wedge \\
(v \stackrel{c}{\Vdash} r_1 : r_2 \ = \ \exists v_1 v_2 l.\ (v = v_1[l]v_2) \wedge v_1[l] \stackrel{c}{\Vdash} r_1 \wedge [l]v_2 \stackrel{c}{\Vdash} r_2) \\
\wedge \\
(v \stackrel{c}{\Vdash} r_1 \mid r_2 \ = \ v \stackrel{c}{\Vdash} r_1 \vee v \stackrel{c}{\Vdash} r_2) \\
\wedge
\end{array}
$$

$$
\begin{array}{l}
(v \models^{c} r_1 \&\& r_2 \;=\; v \models^{c} r_1 \wedge v \models^{c} r_2) \\
\wedge \\
(v \models^{c} [*0] \;=\; (v = \epsilon)) \\
\wedge \\
(v \models^{c} r[*] \;=\; v \models^{c} [*0] \vee \exists v_1\, v_2.\ \neg(v = \epsilon) \wedge (v = v_1\, v_2) \wedge v_1 \models^{c} r \wedge v_2 \models^{c} r[*]) \\
\wedge \\
(v \models^{c} r@c_1 \;=\; v \models^{c_1} r)
\end{array}
$$

The raw HOL input is

```
(S_SEM v c (S_BOOL b) = CLOCK_TICK v c /\ B_SEM (ELEM v (LENGTH v - 1)) b)
/\
(S_SEM v c (S_CAT(r1,r2)) = ?v1 v2. (v = v1 <> v2) /\ S_SEM v1 c r1 /\ S_SEM v2 c r2)
/\
(S_SEM v c (S_FUSION(r1,r2)) =
  ?v1 v2 l. (v = v1 <> [l] <> v2) /\ S_SEM (v1<>[l]) c r1 /\ S_SEM ([l]<>v2) c r2)
/\
(S_SEM v c (S_OR(r1,r2)) = S_SEM v c r1 \/ S_SEM v c r2)
/\
(S_SEM v c (S_AND(r1,r2)) = S_SEM v c r1 /\ S_SEM v c r2)
/\
(S_SEM v c S_EMPTY = (v = []))
/\
(S_SEM v c (S_REPEAT r) =
  S_SEM v c S_EMPTY
  \/ ?v1 v2. ~(v=[]) /\ (v = v1 <> v2) /\ S_SEM v1 c r /\ S_SEM v2 c (S_REPEAT r))
/\
(S_SEM v c (S_CLOCK(r,c1)) = S_SEM v c1 r)
```

4.3. Foundation Language (FL)

FL combines standard LTL notation with a less standard abort operation and some constructs using SEREs. The abstract syntax from B.1 of the LRM is:

The unclocked semantics from B.2.1.1.2 of the LRM is:

We refer to a formula of FL with no @ operator as an *unclocked formula*. Let v be a finite or infinite word, b be a boolean expression, r, r_1, r_2 unclocked SEREs, and φ, ψ unclocked FL formulas. We use $\models$ to define the semantics of unclocked FL formulas: If $v \models \varphi$ we say that v *models* (or *satisfies*) φ.

1. $v \models (\varphi) \iff v \models \varphi$
2. $v \models \neg\varphi \iff \overline{v} \not\models \varphi$
3. $v \models \varphi \wedge \psi \iff v \models \varphi$ and $v \models \psi$
4. $v \models b! \iff |v| > 0$ and $v^0 \Vdash b$
5. $v \models b \iff |v| = 0$ or $v^0 \Vdash b$
6. $v \models r! \iff \exists j < |v|$ s.t. $v^{0..j} \models r$
7. $v \models r \iff \forall j < |v|$, $v^{0..j}\top^\omega \models r!$
8. $v \models X!\, \varphi \iff |v| > 1$ and $v^{1..} \models \varphi$
9. $v \models [\varphi U \psi] \iff \exists k < |v|$ s.t. $v^{k..} \models \psi$, and $\forall j < k,\ \ v^{j..} \models \varphi$
10. $v \models \varphi$ abort $b \iff$ either $v \models \varphi$ or $\exists j < |v|$ s.t. $v^j \Vdash b$ and $v^{0..j-1}\top^\omega \models \varphi$
11. $v \models r \mapsto \varphi \iff \forall j < |v|$ s.t. $\overline{v}^{0..j} \models r$, $v^{j..} \models \varphi$

The pretty-printed HOL version of this is:

$$
\begin{array}{l}
(v \models \neg f \;=\; \neg(\overline{v} \models f)) \\
\wedge \\
(v \models f_1 \wedge f_2 \;=\; v \models f_1 \wedge v \models f_2) \\
\wedge \\
(v \models b! \;=\; (|v| > 0) \wedge v^0 \Vdash b) \\
\wedge \\
(v \models b \;=\; (|v| = 0) \vee v^0 \Vdash b) \\
\wedge \\
(v \models r! \;=\; \exists j \in [0..|v|).\ v^{0..j} \models\!\models r) \\
\wedge \\
(v \models r \;=\; \forall j \in [0..|v|).\ v^{0..j}\top^\omega \models r!) \\
\wedge \\
(v \models X!\, f \;=\; |v| > 1 \wedge v^{1..} \models f) \\
\wedge \\
(v \models [f_1\ U\ f_2] \;=\; \exists k \in [0..|v|).\ v^{k..} \models f_2 \wedge \forall j \in [0..k).\ v^{j..} \models f_1) \\
\wedge \\
(v \models f \ \mathsf{abort}\ b \;=\; v \models f \vee \exists j \in [0..|v|).\ v^j \Vdash b \wedge v^{0..j-1}\top^\omega \models f) \\
\wedge \\
(v \models r \mapsto f \;=\; \forall j \in [0..|v|).\ \overline{v}^{0..j} \models\!\models r \Rightarrow v^{j..} \models f)
\end{array}
$$

The raw HOL is

```
(UF_SEM v (F_NOT f) = ~(UF_SEM (COMPLEMENT v) f))
/\
(UF_SEM v (F_AND(f1,f2)) = UF_SEM v f1 /\ UF_SEM v f2)
/\
(UF_SEM v (F_STRONG_BOOL b) = (LENGTH v > 0) /\ B_SEM (ELEM v 0) b)
/\
(UF_SEM v (F_WEAK_BOOL b) = (LENGTH v = XNUM 0) \/ B_SEM (ELEM v 0) b)
/\
(UF_SEM v (F_STRONG_SERE r) = ?j :: LESS(LENGTH v). US_SEM (SEL v (0,j)) r)
/\
(UF_SEM v (F_WEAK_SERE r) =
  !j :: LESS(LENGTH v).
   UF_SEM (CAT(SEL v (0,j),TOP_OMEGA)) (F_STRONG_SERE r))
/\
(UF_SEM v (F_NEXT f) = LENGTH v > 1 /\ UF_SEM (RESTN v 1) f)
/\
(UF_SEM v (F_UNTIL(f1,f2)) =
  ?k :: LESS(LENGTH v).
    UF_SEM (RESTN v k) f2 /\ !j :: LESS k. UF_SEM (RESTN v j) f1)
/\
(UF_SEM v (F_ABORT (f,b)) =
  UF_SEM v f
  \/
  ?j :: LESS(LENGTH v).
     B_SEM (ELEM v j) b /\ UF_SEM (CAT(SEL v (0,j-1),TOP_OMEGA)) f)
/\
(UF_SEM v (F_SUFFIX_IMP(r,f)) =
  !j :: LESS(LENGTH v).
    US_SEM (SEL (COMPLEMENT v) (0,j)) r ==> UF_SEM (RESTN v j) f)
```

The clocked semantics from B.2.1.2.2 of the LRM is:

The semantics of (clocked) FL formulas is defined with respect to finite/infinite words over Σ and a boolean expression c which serves as the clock context. Let v be a finite or infinite word, b, c, c_1 boolean expressions, r, r_1, r_2 SEREs, and φ, ψ FL formulas. We use $\models^c$ to define the semantics of FL formulas. If $v \models^c \varphi$ we say that v *models* (or *satisfies*) φ *in the context of clock* c.

1. $v \models^c (\varphi) \iff v \models^c \varphi$
2. $v \models^c \neg\varphi \iff \overline{v} \not\models^c \varphi$
3. $v \models^c \varphi \wedge \psi \iff v \models^c \varphi$ and $v \models^c \psi$
4. $v \models^c b! \iff \exists j < |v|$ s.t. $v^{0..j}$ is a clock tick of c and $v^j \Vdash b$
5. $v \models^c b \iff \forall j < |v|$ s.t. $\overline{v}^{0..j}$ is a clock tick of c, $v^j \Vdash b$
6. $v \models^c r! \iff \exists j < |v|$ s.t. $v^{0..j} \not\models^c r$
7. $v \models^c r \iff \forall j < |v|$, $v^{0..j}\top^\omega \not\models^c r!$
8. $v \models^c X!\ f \iff \exists j < k < |v|$ s.t. $v^{0..j}$ and $v^{j+1..k}$ are clock ticks of c and $v^{k..} \models^c f$
9. $v \models^c [\varphi U \psi] \iff \exists k < |v|$ s.t. $v^k \Vdash c$, $v^{k..} \models^c \psi$, and $\forall j < k$ s.t. $\overline{v}^j \Vdash c$, $v^{j..} \models^c \varphi$
10. $v \models^c \varphi$ abort $b \iff$ either $v \models^c \varphi$ or $\exists j < |v|$ s.t. $v^j \Vdash b$ and $v^{0..j-1}\top^\omega \models^c \varphi$
11. $v \models^c r \mapsto \varphi \iff \forall j < |v|$ s.t. $\overline{v}^{0..j} \not\models^c r$, $v^{j..} \models^c \varphi$
12. $v \models^c \varphi @ c_1 \iff v \models^{c_1} \varphi$

The HOL semantics is specified by defining a semantic function `F_SEM` such that `F_SEM` w c f means FL formula f is true of path w with current clock c.
The HOL term `F_SEM` v c f is pretty printed as $v \models^c f$.

$$
\begin{array}{l}
(v \models^c \neg f \ = \ \neg(\overline{v} \models^c f)) \\
\wedge \\
(v \models^c f_1 \wedge f_2 \ = \ v \models^c f_1 \wedge v \models^c f_2) \\
\wedge \\
(v \models^c b! \ = \ \exists j \in [0..|v|).\ \mathsf{ClockTick}(v^{0..j}, c) \wedge v^j \Vdash b) \\
\wedge \\
(v \models^c b \ = \ \forall j \in [0..|v|).\ \mathsf{ClockTick}(\overline{v}^{0..j}, c) \Rightarrow v^j \Vdash b) \\
\wedge \\
(v \models^c r! \ = \ \exists j \in [0..|v|).\ v^{0..j} \not\models^c r) \\
\wedge \\
(v \models^c r \ = \ \forall j \in [0..|v|).\ v^{0..j}\top^\omega \not\models^c r!) \\
\wedge \\
(v \models^c X!\, f \ = \ \exists jk \in [0..|v|).\ j < k \wedge \mathsf{ClockTick}(v^{0..j}, c) \wedge \mathsf{ClockTick}(v^{j+1..k}, c) \wedge v^{k..} \models^c f) \\
\wedge \\
(v \models^c [f_1\ U\ f_2] \ = \ \exists k \in [0..|v|).\ v^k \Vdash c \wedge v^{k..} \not\models^c f_2 \wedge \forall j \in [0..k).\ \overline{v}^j \Vdash c \Rightarrow v^{j..} \models^c f_1) \\
\wedge \\
(v \models^c f \ \mathsf{abort}\ b \ = \ v \models^c f \vee \exists j \in [0..|v|).\ v^j \Vdash b \wedge v^{0..j-1}\top^\omega \models^c f) \\
\wedge \\
(v \models^c f @ c_1 \ = \ v \models^{c_1} f) \\
\wedge \\
(v \models^c r \mapsto f \ = \ \forall j \in [0..|v|).\overline{v}^{0..j} \not\models^c r \Rightarrow v^{j..} \models^c f)
\end{array}
$$

The raw HOL is:

```
(F_SEM v c (F_NOT f) = ~(F_SEM (COMPLEMENT v) c f))
/\
(F_SEM v c (F_AND(f1,f2)) = F_SEM v c f1 /\ F_SEM v c f2)
/\
(F_SEM v c (F_STRONG_BOOL b) =
  ?j :: LESS(LENGTH v). CLOCK_TICK (SEL v (0,j)) c /\ B_SEM (ELEM v j) b)
/\
(F_SEM v c (F_WEAK_BOOL b) =
  !j :: LESS(LENGTH v). CLOCK_TICK (SEL (COMPLEMENT v) (0,j)) c ==> B_SEM (ELEM v j) b)
/\
(F_SEM v c (F_STRONG_SERE r) = ?j :: LESS(LENGTH v). S_SEM (SEL v (0,j)) c r)
/\
(F_SEM v c (F_WEAK_SERE r) =
  !j :: LESS(LENGTH v). F_SEM (CAT(SEL v (0,j),TOP_OMEGA)) c (F_STRONG_SERE r))
/\
(F_SEM v c (F_NEXT f) =
  ?j k :: LESS(LENGTH v).
    j < k                          /\
    CLOCK_TICK (SEL v (0,j)) c     /\
    CLOCK_TICK (SEL v (j+1,k)) c /\
    F_SEM (RESTN v k) c f)
/\
(F_SEM v c (F_UNTIL(f1,f2)) =
  ?k :: LESS(LENGTH v).
    B_SEM (ELEM v k) c /\
    F_SEM (RESTN v k) c f2 /\
    !j :: LESS k. B_SEM (ELEM (COMPLEMENT v) j) c ==> F_SEM (RESTN v j) c f1)
/\
(F_SEM v c (F_ABORT (f,b)) =
  F_SEM v c f
  \/
  ?j :: LESS(LENGTH v). B_SEM (ELEM v j) b /\ F_SEM (CAT(SEL v (0,j-1),TOP_OMEGA)) c f)
/\
(F_SEM v c (F_CLOCK(f,c1)) = F_SEM v c1 f)
/\
(F_SEM v c (F_SUFFIX_IMP(r,f)) =
  !j :: LESS(LENGTH v). S_SEM (SEL (COMPLEMENT v) (0,j)) c r ==> F_SEM (RESTN v j) c f)
```

5. Definitions and proofs

The HOL versions of the semantics given in the preceding sections were not the actual definitions of the semantic functions `US_SEM`, `S_SEM`, `UF_SEM` and `F_SEM`, but were theorems derived from reformulations of the LRM definitions to make them fall within the scope of the HOL definitional tools provided by the TFL package [Sli96]. Definitions in HOL simply declare of a name for an existing closed term. Recursive 'definitions' are made by compiling equations into primitive definitions (using recursion theorems), making the definition using HOL's definition mechanism, and then deriving the equation one wants. For simple recursive equations this is handled completely automatically by TFL. For recursions that are not simple there are two options: (i) supply a proof script when making the definition (which typically involves giving some well-founded relation that ensures the recursion terminates on all arguments), or (ii) first defining a simple recursion and then deducing the desired 'definitional' equation as a theorem. We used approach (ii) for the PSL 1.1 semantics (approach (i) was used with the 1.01 semantics).

As an example, consider the definition of the unclocked semantics of the repetition SERE r`[*]` (the same issue arises with the clocked semantics). The definition of $v \models r$ is mostly by a structural recursion on the syntax of SEREs r. However, the clause defining $v \models r[*]$ does not recurse on r, but instead on v:

$$v \models r[*] \;=\; v \models [*0] \vee \exists v_1\, v_2.\ \neg(v = \epsilon) \wedge (v = v_1\, v_2) \wedge v_1 \models r \wedge v_2 \models r[*]$$

Observe that $v_2 \models r[*]$ occurs in the right hand side of the equation. TFL cannot automatically prove that this LRM semantics is well-founded.

The actual definition used in HOL for the r`[*]` case does recurse on r and is:

$$v \models r[*] \;=\; \exists vlist.\ (v = \mathsf{Concat}\ vlist) \wedge \mathsf{All}(\lambda v'.v' \models r)vlist$$

where Concat *vlist* concatenates (flattens) a list of lists and All P *vlist* applies a predicate P to each member of *vlist* and conjoins the results (i.e. combines the results with $\wedge$). The LRM equation is then deduced from the definition with Concat and All

In both the FAC and CHARME papers we described theorems about the semantics that had been mechanically proved using the HOL system. These were either 'sanity checking' properties that helped validate the semantics (FAC paper), or reformulations of the semantics needed to support tools that worked by deduction (CHARME paper).

So far we have only proved a few properties of the 1.1 semantics. These are of the 'sanity checking' kind and are taken from the first page of an unpublished paper entitled *Some characteristics of Accellera PSL* by Cindy Eisner, Dana Fisman and John Havlicek. The lemmas proved in HOL so far are all about SEREs:

$$\vdash\ \mathsf{ClockTick}(v, \mathrm{T}) \;=\; \exists kl.\ \neg(l = \bot) \wedge (v = \top^k[l])$$

$$\vdash\ \forall rvc.\ |v| > 0 \wedge \mathsf{ClockFree}(r) \wedge v \overset{c}{\models} r \Rightarrow v^{|v|-1} \Vdash c$$

$$\vdash\ \forall r.\ \mathsf{ClockFree}(r) \Rightarrow \forall v.\ v \models r[+] \;=\; \exists vlist.\ (v = \mathsf{Concat}\ vlist) \wedge |vlist| > 0 \wedge \mathsf{All}(\lambda v.v \models r)vlist$$

$$\vdash\ \forall rcv.\ v \overset{c}{\models} r[+] \;=\; \exists vlist.\ (v = \mathsf{Concat}\ vlist) \wedge |vlist| > 0 \wedge \mathsf{All}(\lambda v.v \overset{c}{\models} r)vlist$$

$$\vdash\ \forall r.\ \mathsf{ClockFree}(r) \Rightarrow \forall v.v \models r \Rightarrow \mathsf{BottomFree}(v)$$

$$\vdash\ \forall rcv.\ v \overset{c}{\models} r \Rightarrow \mathsf{BottomFree}(v)$$

$$\vdash\ \forall rv.\ \mathsf{ClockFree}(r) \wedge v \models r \Rightarrow \forall k \in [0..|v|).\ v^{0..k}\top^{(|v|-k-1)} \models r$$

In these lemmas, ClockFree(r) is defined to mean that r has no sub-term containing `@` (i.e. is in the unclocked subset), BottomFree(v) is defined to mean that no letter of v is $\bot$ and $r[+]$ is syntactic sugar for $r; r[*]$ (which in raw HOL is `S_CAT(`r`,S_REPEAT` r`)`). All these lemmas were routine to prove (though the r`[*]` case of the last lemma was surprisingly tedious).

The representation of these lemmas in raw HOL is:

```
|- CLOCK_TICK v B_TRUE = ?k l. ~(l = BOTTOM) /\ (v = TOP_ITER k <> [l])

|- !r v c.
     LENGTH v > 0 /\ S_CLOCK_FREE r /\ S_SEM v c r ==>
     B_SEM (ELEM v (LENGTH v - 1)) c

|- !r.
     S_CLOCK_FREE r ==>
     !v.
       US_SEM v (S_NON_ZERO_REPEAT r) =
       ?vlist.
         (v = CONCAT vlist) /\ LENGTH vlist > 0 /\
         ALL_EL (\v. US_SEM v r) vlist

|- !r c v.
     S_SEM v c (S_NON_ZERO_REPEAT r) =
     ?vlist.
       (v = CONCAT vlist) /\ LENGTH vlist > 0 /\
       ALL_EL (\v. S_SEM v c r) vlist
```

```
|- !r. S_CLOCK_FREE r ==> !v. US_SEM v r ==> BOTTOM_FREE v

|- !r c v. S_SEM v c r ==> BOTTOM_FREE v

|- !r v.
     S_CLOCK_FREE r /\ US_SEM v r ==>
     !k::LESS (LENGTH v).
       US_SEM (SEL v (0,k) <> TOP_ITER (LENGTH v - k - 1)) r
```

We hope to prove more properties about SEREs and also some properties about formulas. In particular, validating rewrites that translate clocked to unclocked SEREs and formulas is necessary to support our tools based on the semantics.

6. Acknowledgements

The work described here would not have been possible without the help of Cindy Eisner and Dana Fisman from the PSL/Sugar team at IBM. Dana Fisman supplied the LaTeX sources for the extracts of the draft Version 1.1 LRM shown in the framed boxes. Keith Wansbrough's HOL-to-LaTeX tool for typesetting the ASCII syntax of HOL terms was invaluable for generating the pretty-printed versions of the HOL semantics. This work was supported by an IBM Faculty Award.

References

[GHS03] Mike Gordon, Joe Hurd, and Konrad Slind. Executing the formal semantics of the Accellera Property Specification Language by mechanised theorem proving. In Daniel Geist and Enrico Tronci, editors, *Proc.* 12^{th} *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2003)*, Lecture Notes in Computer Science. Springer-Verlag, October 2003. 21 - 24 October 2003, University of L'Aquila, Computer Science Department, L'Aquila, Italy.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem-proving environment for higher-order logic.* Cambridge University Press, 1993.

[Gor03] Michael J. C. Gordon. Validating the PSL/Sugar Semantics Using Automated Reasoning. *Formal Aspects of Computing*, 15(4):406–421, 2003.

[Sli96] K. Slind. Function definition in higher order logic. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, Turku, Finland, August 1996: Proceedings*, volume 1125 of *Lecture Notes in Computer Science*, pages 381–397. Springer-Verlag, 1996.