A Suite of Hard ACL2 Theorems Arising in Refinement-Based Processor Verification

Panagiotis Manolios[1] and Sudarshan K. Srinivasan[2]

[1] Georgia Institute of Technology
College of Computing, CERCS
[2] School of Electrical & Computer Engineering
Atlanta, GA 30332-0280
`manolios@cc.gatech.edu, darshan@ece.gatech.edu`
`http://www.cc.gatech.edu/∼manolios`
`http://www.ece.gatech.edu/∼darshan`

Abstract. We have been using ACL2 to verify pipelined machine models for several years and have compiled a suite of 18 problems that arose in the theorem proving process. We believe that this suite will be useful for the future development of ACL2 because it consists of difficult problems that arise in practice, and furthermore, these problems can be handled efficiently by other methods. For example, ACL2 was able to prove the simplest problem in the suite after $15\frac{1}{2}$ days, but UCLID was able to prove the same theorem in seconds.

1 Introduction

We have compiled a suite of 18 theorems arising in refinement-based proofs of correctness for term-level pipelined machine models in ACL2. The simplest of these problems is a correctness theorem for a 5-stage DLX-like pipelined machine, which takes ACL2 about $15\frac{1}{2}$ days to prove. The other problems are significantly more complex.

While ACL2 [7, 6] has been successfully applied to a wide range of commercially interesting hardware verification problems (*e.g.*, [14, 17, 16, 2, 5, 4]), our suite identifies a class of problems, naturally arising in practice, that ACL2 has extreme difficulty handling, but which can be easily handled by existing tools, *e.g.*, UCLID [8, 9]. Our hope is that our suite of problems will stimulate research on improving ACL2's reasoning abilities.

The rest of the paper is organized as follows. In Section 2, we give an overview of refinement-based processor verification, the domain from which the suite of problems originates. In Section 3, we give an example of a refinement theorem in ACL2. In Section 4, we review the UCLID decision procedure, and in Section 5 we give a detailed description of the suite of problems. In Section 6, we outline an approach to improving ACL2's reasoning abilities that we are currently working on, and we conclude with Section 7.

2 Refinement Based Processor Verification

In this section, we describe the theory of refinement that our microprocessor correctness proofs are based on. For the full details see [11, 12, 10].

The point of a correctness proof is to establish a meaningful relationship between ISA, a machine modeled at the instruction set architecture level and MA, a machine modeled at the microarchitecture level, a low level description that includes the pipeline. We accomplish this by first defining a *refinement map*, r, a function from MA states to ISA states; think of r as showing us how to view an MA state as an ISA state. We then prove a *stuttering bisimulation refinement*: for every pair of states w, s such that w is an MA state and $s = r(w)$, for every infinite path σ starting at s, there is a "matching" infinite path δ starting at w, and conversely. That σ and δ match implies that applying r to the states in δ results in a sequence that can be obtained from σ by repeating, but only finitely often, some of σ's states, as MA may require several steps before matching a single step of ISA. A problem with this approach is that it requires reasoning about infinite paths, which is difficult to automate. In [11], we give an equivalent formulation, WEB-refinement, that requires only local reasoning. We now give the relevant definitions, which are given in terms of general transition systems (TS). A TS $\mathcal{M}$ is a triple $\langle S, \dashrightarrow, L\rangle$, consisting of a set of states, S, a transition relation, $\dashrightarrow$, and a labeling function L with domain S, where $L(s)$ is what is visible at s.

Definition 1. *(WEB Refinement) Let $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$, $\mathcal{M}' = \langle S', \dashrightarrow', L'\rangle$, and $r : S \to S'$. We say that $\mathcal{M}$ is a WEB refinement of $\mathcal{M}'$ with respect to refinement map r, written $\mathcal{M} \approx_r \mathcal{M}'$, if there exists a relation, B, such that $\langle \forall s \in S :: sBr(s)\rangle$ and B is a WEB on the TS $\langle S \uplus S', \dashrightarrow \uplus \dashrightarrow', \mathcal{L}\rangle$, where $\mathcal{L}(s) = L'(s)$ for s an S' state and $\mathcal{L}(s) = L'(r(s))$ otherwise.*

In the above definition, it helps to think of $\mathcal{M}'$ as corresponding to ISA and $\mathcal{M}$ as corresponding to MA. Note that in the disjoint union of $\mathcal{M}$ and $\mathcal{M}'$, the label of every $\mathcal{M}$ state, s, matches the label of the corresponding $\mathcal{M}'$ state, $r(s)$. WEBs are defined next; the main property enjoyed by a WEB, say B, is that all states related by B have the same (up to stuttering) visible behaviors.

Definition 2. *$B \subseteq S \times S$ is a WEB on TS $\mathcal{M} = \langle S, \dashrightarrow, L\rangle$ iff:*

(1) B is an equivalence relation on S; and

(2) $\langle \forall s, w \in S :: sBw \;\Rightarrow\; L(s) = L(w)\rangle$; and

(3) There exist functions $erankl : S \times S \to \mathbb{N}$, $erankt : S \to W$, such that $\langle W, \lessdot\rangle$ is well-founded, and

$\langle \forall s, u, w \in S :: sBw \;\wedge\; s \dashrightarrow u \;\Rightarrow$

(a) $\langle \exists v :: w \dashrightarrow v \;\wedge\; uBv\rangle \;\vee$

(b) $(uBw \;\wedge\; erankt(u) \lessdot erankt(s)) \;\vee$

(c) $\langle \exists v :: w \dashrightarrow v \;\wedge\; sBv \;\wedge\; erankl(v, u) < erankl(w, u)\rangle\rangle$

The third WEB condition says that given states s and w in the same class, such that s can step to u, u is either matched by a step from w, or u and w are in the same class and a rank function decreases (to guarantee that w is eventually forced to take a step), or some successor v of w is in the same class as s and a rank function decreases (to guarantee that u is eventually matched). To prove that a relation is a WEB, reasoning about single steps of $\dashrightarrow$ suffices.

The refinement theorem contains quantifiers and involves exhibiting the existence of certain rank functions. We would prefer to reduce the proof obligation to a decidable fragment of first-order logic, which we do as follows. First, we strengthen the refinement theorem in such a way that it can be reduced to a formula expressible in CLU (the logic of Counter arithmetic with Lambda expressions and Uninterpreted functions). Second, we show how to define rank functions in a general way that does not require deep understanding of the pipelined machine.

To strengthen the WEB-refinement proof obligation so that we obtain a CLU expressible statement, we start by defining the equivalence classes of B to consist of one ISA state and all the MA states that map to the ISA state under r. Now, condition 2 of the WEB definition clearly holds. Our ISA and MA machines are deterministic (actually they are nondeterministic, but we use oracle variables to make them deterministic [12]), thus, after some symbolic manipulation, we can strengthen condition 3 of the WEB definition to the following "core theorem", where $rank$ is a function that maps states of MA into the natural numbers.

$$\begin{aligned} \langle \forall w \in \texttt{MA} :: \quad & s = r(w) \;\wedge\; u = \texttt{ISA-step}(s) \;\wedge \\ & v = \texttt{MA-step}(w) \;\wedge\; u \neq r(v) \\ \Longrightarrow \quad & s = r(v) \;\wedge\; rank(v) < rank(w) \rangle \end{aligned}$$

In the formula above s and u are ISA states, and w and v are MA states; `ISA-step` is a function corresponding to stepping the ISA machine once and `MA-step` is a function corresponding to stepping the MA machine once. The core theorem says that if w refines s, *u* is obtained by stepping s, v is obtained by stepping w, and v does not refine u, then v refines s and the $rank$ of v is less than the $rank$ of w. The proof obligation relating s and v is the safety component, and the proof obligation that $rank(v) < rank(w)$ is the liveness component.

We use two types of refinement maps. One is based on flushing, where partially executed instructions in the microprocessor state are completed without fetching any new instructions. Thus, all the pipeline latches are invalidated, giving rise to an instruction architecture state. The other is based on the commitment approach that can be loosely thought of as the dual of flushing, since partially completed instructions are invalidated instead of being completed.

3 Refinement Theorems in ACL2

In this section we examine the ACL2 core WEB-refinement theorem for the 5-stage pipelined model mentioned previously.

The refinement theorem has three parts: one describes the microprocessor model, one describes the instruction set architecture model, and one contains the actual theorem that relates these models. The models are at the term-level, *i.e.,* the data path is abstracted away, as is combinational circuitry such as the ALU (using encapsulate).

Recall that the term-level refinement theorem is specified in a decidable fragment of first-order logic. The complexity of the refinement theorem arises from defining refinement maps that map microprocessor states to instruction set architecture states and from invariants that characterize the set of reachable states. The refinement maps and invariants are defined using sequences of symbolic simulation steps of the microprocessor model, and for brevity, are not shown.

The ACL2 code fragment below shows part of the refinement theorem for the 5-stage DLX pipeline. The implementation and specification states are initialized with arbitrary constants using the `initialize` function. The types of these constants are defined in the antecedent of the implication. Then, the implementation and specification states are "stepped" using the `simulate` function. This example uses the refinement map based on the commitment approach and requires the use of invariants such as `Good_MA`. Since the theorem also checks liveness, we see that a rank function is used in the consequent.

```
(defthm WEB_CORE
  (implies
   (and
    (integerp fdpPC0)
    (integerp depPC0)
    (booleanp deRegWrite0)
    ...
    )
   (let* ((ST0 (initialize fdpPC0 depPC0 ...))
          (ST1 (simulate ST0 nil pc0 nil nil pc0
                         (g 'pRF (g 'impl ST0))
                         (g 'pDMemHist_1 (g 'impl ST0))))
          (ST2 (simulate ST1 nil pc0 nil nil pc0
                         (g 'pRF (g 'impl ST1))
                         (g 'pDMemHist_1 (g 'impl ST1))))
          ...
          (Good_MA_V (Good_MA_a
                      Equiv_MA_0
                      Equiv_MA_1
                      Equiv_MA_2
                      Equiv_MA_3
                      Equiv_MA_4))
          ...
          (Rank_V (rank_a (g 'mwWRT (g 'impl ST34))
                          (g 'emWRT (g 'impl ST34))
                          (g 'deWRT (g 'impl ST34))
                          (g 'fdWRT (g 'impl ST34))
                          ZERO))
          (S_pc1 (g 'sPC (g 'speci ST35)))
          (S_rf1 (g 'sRF (g 'speci ST35)))
          (S_dmem1 (g 'sDMem (g 'speci ST35))))
     (and
      Good_MA_V
      (or
       (not
        (and
         (equal S_pc0 I_pc0)
         (equal (read-sRF_a a S_rf0) (read-pRF_a a I_rf0))
         (equal S_dmem0 I_dmem0)))
       (or
        (and
         (equal S_pc1 I_pc)
         (equal (read-sRF_a a S_rf1) (read-pRF_a a I_rf))
         (equal S_dmem1 I_dmem))
        (and
         (equal S_pc0 I_pc)
         (equal (read-sRF_a a S_rf0) (read-pRF_a a I_rf))
         (equal S_dmem0 I_dmem)
         (< Rank_V Rank_W))))
      ...))))
```

4 UCLID System

UCLID [3, 9] is a decision procedure for formulas expressed in a decidable fragment of first order logic called CLU. The CLU logic contains Booleans connectives, uninterpreted functions, equality, counter arithmetic, ordering, and restricted lambda expressions. Terms of the logic are used to abstract word-level values and uninterpreted functions are used to abstract combinational circuit blocks. Uninterpreted functions only satisfy the property of functional consistency, *i.e.*, if the inputs of two different instances of an uninterpreted function are equal, then their outputs are equal.

The UCLID specification language is used to model processors and to specify the correctness formulas. The processor models in UCLID are specified at the term level. A symbolic simulation engine that is part of UCLID takes the processor model and the correctness formula as input, and generates the corresponding CLU formula, which is then translated to a propositional formula. The translation process takes advantage of innovative encoding techniques. The propositional formula is checked using a state-of-the-art SAT solvers such as Chaff [15] and Siege [18]. The UCLID system has been used to verify out-of-order microprocessors models at the term level [8].

Using UCLID, we were able to prove the core refinement theorem of the 5-stage pipeline example in about 3 seconds. Note that the UCLID system does not have enough expressive power to state the full correctness theorem, but ACL2 was able to complete the rest of the proof in under a minute.

5 Suite of Theorems

The problem suite is a set of refinement theorems for processor models in ACL2. As we have stated earlier, the simplest of these refinement theorems took ACL2 $15\frac{1}{2}$ days to prove. The other benchmarks are an order of magnitude harder. An interesting feature of the suite is that the ACL2 theorems are restricted to a decidable fragment of first order logic. We used the UCLID decision procedure to prove the theorems and report the verification times below, but first we give a detailed description of each benchmark.

5S, 5S-Part: The '5S' benchmark is a refinement theorem for a 5-stage pipelined machine model with register-register, register-immediate, and store instruction types. The pipelined machine model is similar to the DLX pipeline. The refinement map is based on the commitment approach. The 5S-Part benchmark is part of the correctness proof described in 5S. We ran 5S-Part to completion using ACL2 and the proof took $15\frac{1}{2}$ days to complete. When ACL2 was configured to suppress printing, the proof took 10 days.

CXS: This benchmark is a refinement theorem for a 7 stage pipelined machine inspired by the Intel XScale architecture. It has 5 abstract instruction types including register-register, register-immediate, branch, loads, and stores. The refinement map used is based on the commitment approach. The benchmark includes ACL2 models of both the 7 stage pipelined machine (implementation model) and its instruction set architecture (specification model).

CXS-BP: This benchmark is an extension of the CXS benchmark. It is obtained by adding branch prediction to the 7 stage pipelined machine model. The refinement map is based on the commitment approach and is modified from CXS to accommodate branch prediction.

CXS-BP-EX: This benchmark is obtained by including exceptions to CXS-BP implementation model. The refinement map is again based on the commitment approach. The specification model and the refinement map are modified to incorporate exceptions.

CXS-BP-EX-INP: This benchmark adds interrupts to CXS-BP-EX. The refinement map is based on the commitment approach and makes use of oracle variables to deal with the interrupts.

FXS, FXS-BP, FXS-BP-EX, FXS-BP-EX-INP: The benchmarks FXS, FXS-BP, FXS-BP-EX, FXS-BP-EX-INP are similar to CXS, CXS-BP, CXS-BP-EX, and CXS-BP-EX-INP, respectively, in that they are refinement proofs for the same implementation models. The main difference is that these benchmarks use flushing as a refinement map.

Table 1 lists the problems and also provides the time taken by the UCLID decision procedure to prove the theorems. UCLID was run with the Siege SAT solver, and we also report statistics for the CNF formulas produced by UCLID. The UCLID results are based on our previous work [13]. The suffix "-S" indicates that the theorem is a safety theorem, while the suffix "-SL" indicates that the theorem is the full core theorem, containing both the safety and liveness components.

The total column in Table 1 is the sum of the time taken for running UCLID and Siege. We would like to point out here that only the 5S-Part problem was run to completion in ACL2. The other problems are an order of magnitude harder, as can be seen from the UCLID verification times. The ACL2 times for the other problems are therefore extrapolated values and are shown in italics.

Table 1 Verification times and CNF statistics for the benchmark suite using UCLID and Siege

Benchmark	CNF Vars	CNF Clauses	UCLID [sec]			ACL2 [sec]
			UCLID	Siege	Total	
5S-Part	5,285	15,457	1	2	3	1,339,200
5S	5,285	15,457	1	2	3	*1,339,200*
CXS-S	12,930	38,215	3	35	38	*16,963,200*
CXS-SL	12,495	36,925	3	29	32	*14,284,800*
CXS-BP-S	24,640	72,859	5	284	289	*129,009,600*
CXS-BP-SL	23,913	70,693	5	300	305	*136,152,000*
CXS-BP-EX-S	24,651	72,841	5	244	249	*111,153,600*
CXS-BP-EX-SL	24,149	71,350	5	233	238	*106,243,200*
CXS-BP-EX-INP-S	24,669	72,880	6	255	261	*116,510,400*
CXS-BP-EX-INP-SL	24,478	72,322	6	263	269	*120,081,600*
FXS-S	28,505	84,619	14	140	154	*68,745,600*
FXS-SL	53,441	159,010	15	160	175	*78,120,000*
FXS-BP-S	33,964	100,624	15	170	185	*82,584,000*
FXS-BP-SL	71,184	211,723	16	187	203	*90,619,200*
FXS-BP-EX-S	35,827	106,114	16	179	195	*87,048,000*
FXS-BP-EX-SL	74,591	221,812	17	163	180	*80,352,000*
FXS-BP-EX-INP-S	38,711	11,4742	19	128	147	*65,620,800*
FXS-BP-EX-INP-SL	781,121	241,345	19	170	189	*84,369,600*

The ACL2 theorems were obtained by translating UCLID specifications to ACL2 with a translator we wrote. There is the danger that the translation is what is responsible for the slow ACL2 verification times. To better understand how the translator affects verification times, we considered the correctness theorem for a 3 stage pipelined machine written for ACL2. After considerable effort was devoted to ACL2 efficiency considerations, ACL2 took 130 seconds to prove the theorem. We then translated (by hand) the theorem to UCLID, which took about 1.7 seconds to complete the proof. We then used our tool to translate the UCLID specification back to ACL2. The resulting theorem took ACL2 430 seconds to prove. Even accounting for the factor of four slowdown, there is still a big gap between the time taken by UCLID and the time taken by ACL2.

6 Integrating Decision Procedures in ACL2

Our suite of problems convincingly shows that UCLID is a useful tool for pipeline machine verification, but why do we need ACL2 and why would we want to integrate UCLID into ACL2? We give three reasons, though there are several other good reasons. First, UCLID models are at the term-level and are not executable. Second, ACL2 is far more expressive than the CLU specification language, which is not expressive enough to even state the WEB refinement theorem, though it can be used to state the “core” theorem. ACL2 allows us to state the WEB theorem and to formally reduce it to the core theorem. Third, ACL2 can be used to model and reason about pipelined machines at various levels of abstraction, including at the term and bit levels. However, the UCLID decision procedure is only suitable for term-level models.

We believe that integrating UCLID (and other similar decision procedures) into ACL2 will result in a system that is more powerful than the sum of its parts. We are currently exploring this possibility. The difficulty with the fine-grained integration of decision procedures into heuristic theorem provers is well-known [1], but we hope to avoid these problems by integrating UCLID in a course-grained way. The idea is to embed the CLU logic into ACL2, something that is not entirely trivial, as it is possible in UCLID to have variables that are assigned lambdas and the ACL2 universe does not contain functions. However, the lambdas used in UCLID are sufficiently restricted that this obstacle can be overcome. Once this embedding is complete, we plan to verify processor models too complex to handle with ACL2 or UCLID alone, but which can be easily handled by our combined system.

7 Conclusion

We have presented a benchmark suite of 18 theorems from the domain of processor verification that ACL2 has difficulty proving. We hope that the suite will help stimulate research on extending ACL2's ability to reason about such problems, and we proposed a first step in this direction.

References

1. R. S. Boyer and J. S. Moore. Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic. In *Machine intelligence 11*, pages 83–124. Oxford University Press, Inc., 1988.
2. B. Brock and W. A. Hunt, Jr. Formally specifying and mechanically verifying programs for the Motorola complex arithmetic processor DSP. In *1997 IEEE International Conference on Computer Design*, pages 31–36. IEEE Computer Society, Oct. 1997.
3. R. E. Bryant, S. K. 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. Larsen, editors, *Computer-Aided Verification–CAV 2002*, volume 2404 of *LNCS*, pages 78–92. Springer-Verlag, 2002.
4. D. A. Greve. Symbolic simulation of the JEM1 microprocessor. In *Formal Methods in Computer-Aided Design – FMCAD*, LNCS. Springer-Verlag, 1998.
5. D. Hardin, M. Wilding, and D. Greve. Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle. In A. J. Hu and M. Y. Vardi, editors, *Computer-Aided Verification – CAV '98*, volume 1427 of *LNCS*. Springer-Verlag, 1998. See URL `http://pobox.com/users/hokie/docs/concept.ps`.
6. M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.
7. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.
8. S. Lahiri, S. Seshia, and R. Bryant. Modeling and verification of out-of-order microprocessors using UCLID. In *Formal Methods in Computer-Aided Design (FMCAD'02)*, volume 2517 of *LNCS*, pages 142–159. Springer-Verlag, 2002.

9. S. K. Lahiri and S. Seshia. The UCLID decision procedure. In *Computer Aided Verification, CAV'04,* volume 3114 of *LNCS*, pages 475–478, July 2004.
10. P. Manolios. Correctness of pipelined machines. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design–FMCAD 2000*, volume 1954 of *LNCS*, pages 161–178. Springer-Verlag, 2000.
11. P. Manolios. *Mechanical Verification of Reactive Systems*. PhD thesis, University of Texas at Austin, August 2001. See URL `http://www.cc.gatech.edu/~manolios/-publications.html`.
12. P. Manolios. A compositional theory of refinement for branching time. In D. Geist and E. Tronci, editors, *12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003*, volume 2860 of *LNCS*, pages 304–318. Springer-Verlag, 2003.
13. P. Manolios and S. Srinivasan. Automatic verification of safety and liveness for XScale-like processor models using WEB-refinements. In *Design Automation and Test in Europe, DATE'04*, LNCS, 2004.
14. J. S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the AMD5$_K$86 floating-point division program. *IEEE Trans. Comp.,* 47(9):913–926, September 1998.
15. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *Design Automation Conference (DAC'01)*, pages 530–535, 2001.
16. D. M. Russinoff. 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. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998.
17. D. M. Russinoff. A mechanically checked proof of correctness of the AMD-K5 floating-point square root microcode. *Formal Methods in System Design*, 14:75–125, 1999.
18. L. Ryan. Siege homepage. See URL `http://www.cs.sfu.ca/ ~loryan/-personal`.