Industrial Strength Refinement Checking

Jesse Bingham*, John Erickson†, Gaurav Singh‡, and Flemming Andersen§
Intel Corporation,
*Email: jesse.d.bingham@intel.com, †Email: john.erickson@intel.com,
‡Email: gaurav.2.singh@intel.com, §Email: flemming.l.andersen@intel.com.

***Abstract*—This paper discusses a methodology used on an industrial hardware development project to validate various cache-coherence protocol components. The idea is to use a high level model (HLM) written in Murphi for model checking purposes, and then to use the HLM as a checker during dynamic (i.e. simulation based-) validation of the RTL. Such a checker requires a formal notion of what it means for the RTL to implement the HLM. Due to RTL pipelining, concurrency, and different RTL/HLM semantics, an appropriate notion is non-obvious. We employ a notion we call *behavioral refinement*, and describe a methodology for creating *refinement checkers*. A novel aspect of our methodology is that all "ingredients" are specified using System Verilog (SV): even the Murphi model itself is compiled into SV. Thus any off-the-shelf SV simulation engine can be used. We report the successful use of our refinement checkers to catch bugs in a real project at Intel and give an example illustrating our methodology.**

I. Introduction

A commonly used approach in verifying and validating complex hardware components involves constructing a *high level model* (HLM) of the component (e.g. [4], [8], [3], [13]). The HLM is a simplified version of the component that still models key functionality. For complex protocols such as cache coherency, the HLM is often written in a nondeterministic guarded-command language, such as Murphi [8]. Model checking small configurations of the HLM is then used to establish correctness of the protocol. Unfortunately, the HLM is too high-level to be synthesized into a high performance pipelined circuit, so a manually-written *register transfer level* (RTL) description is still the primary way of specifying the real hardware. This leaves open the question of whether or not the RTL is consistent with the HLM that was verified.

In this paper, we present a definition of what *implements* means in our context, which is based on the notion of *refinement* [2]. Our notion is tailored to the particular style of HLM (nondeterministic guarded commands) and implementation (clocked RTL), and is interesting in that it allows the HLM to take any finite number of steps per RTL clock; this was necessary both for the toy example given in this paper and the real industrial design we worked on. Many previous papers that allow for this are about super-scalar microprocessor verification, wherein the HLM must execute (up to) the instruction fetch width of the implementation. A few previous works involving HLMs that are similar to ours have identified a need for this allowance [7], [15].

We leave formal verification of this refinement relation as future work, since the techniques are not mature enough to be applied in a predictable way to a real industrial project that is constrained by strict schedules. However, we believe that *if you can't check it, you surely can't verify it*. By *check* we mean to watch for refinement violations during dynamic simulation, a technique we call *refinement checking*. In this light, we pursued refinement checking with the distinct goals of 1) evaluating the level of difficulty in writing the requisite refinement mapping, and 2) catching bugs during RTL development. We concluded that (1) is nontrivial, but certainly not prohibitively difficult. We were also successful at (2) during the few short months between bringing one of our refinement checkers online and writing this paper. We believe this is the first detailed report of using refinement checking on a industrial hardware design project *during development*[1].

One interesting aspect of our methodology is that all ingredients are ultimately written in System Verilog (SV). Hence any off-the-shelf SV simulator can be employed; there is no need to link the simulator with a model checker as was done in previous work [15]. To facilitate this, we developed a translator from Murphi code to SV called *mu2sv*. Mu2sv does a straightforward translation that maps each Murphi rule R to an SV function R`_sv` that take a record of type `MURPHI_STATE` as well as parameters for the rulesets.[2] R`_sv` returns another `MURPHI_STATE` that corresponds to the result of firing the rule.

II. Refinement Checking

Our goal is to monitor the RTL during simulation and flag an error if the behavior is not allowed by the Murphi HLM. In order to do so, we first must define what this means. Following that we present our methodology, and then conclude this section with a detailed example.

In Fig. 2 we pictorially define behavioral refinement, a notion that is relative to a given *refinement map* (RM). A RM is simply a many-to-one function that takes an RTL state and returns an HLM state. Given such a map, the RTL behaviorally refines the HLM if for any RTL behavior (shown across the bottom of the Fig. 2), there exists an HLM behavior that includes all RTL states mapped through the RM as shown. The figure shows that behavioral refinement allows each RTL clock to correspond to 0, 1, or more Murphi rules firing. Implicit in Fig. 2 is the fact that the first RTL state (the first state after reset in practice) must map to an *initial* `MURPHI_STATE`.

[1]The work of Tasiran et al. [15], though compelling, was done after the design was done and didn't catch any bugs.

[2]The `MUPRHI_STATE` SV type declaration is automatically generated by mu2sv and has a field for each variable in the Murphi model

```
type -------- Type Declarations --------------
CacheIndex : 0..1023;
CacheEntry : record
   State : enum {Invalid, Dirty, Clean};
   Addr  : ADDR;
   Data  : DATA;   end;

var ---- State Variable Declarations ---------
CacheArray : array [CacheIndex] of CacheEntry;
Cpu2Cache: Cpu2Cache_t;
Cache2Mem: Cache2Mem_t;

------ Rules (a.k.a Guarded Commands) --------
Ruleset i : CacheIndex "RecvStore"
  (Cpu2Cache.opcode = Store &
   CacheArray[i].State != Invalid &
   CacheArray[i].Addr = Cpu2Cache.Addr) |
  ( (forall j : CacheIndex :
      CacheArray[j].Addr != Cpu2Cache.Addr |
      CacheArray[j].State = Invalid) &
      CacheArray[i].State = Invalid ) ==>
  CacheArray[i].Data := Cpu2Cache.Data;
  CacheArray[i].State := Dirty;
  Absorb(Cpu2Cache);
end

Ruleset i : CacheIndex "Evict"
  CacheArray[i].State != Invalid ==>
  if (CacheArray[i].State == Dirty) begin
    Cache2Mem.opcode := WriteBack;
    Cache2Mem.Addr = CacheArray[i].Addr;
    Cache2Mem.Data = CacheArray[i].Data;
  end;
  CacheArray[i].State := Invalid;
end
```

Fig. 1. Murphi Code for Toy Cache Controller Example

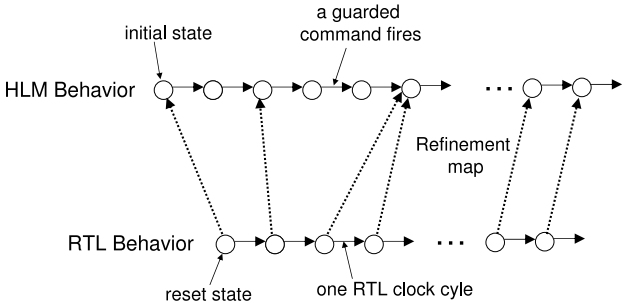


Fig. 2. Behavioral Refinement

We generalize this notion modestly by allowing the refinement function to depend on so-called *history variables* [2]. History variables are auxiliary variables added to the RTL that have no affect on RTL behavior, but rather record information about the past. In practice, history variables are extremely useful for writing a RM for pipelined RTL. We call an RTL state with history variables simply an *augmented RTL state*.

Refinement Checking Methodology. A refinement checker observes an RTL behavior (driven by some test stimuli), and attempts to construct a corresponding Murphi behavior that it behaviorally refines. If it is ever unable to do so, an error is flagged. This requires two distinct "ingredients" to be written by the human: the RM and *rule selection*. In our methodology, both of these are written in SV. Though mathematically `RM()` is a function that takes the current RTL state `r`, as an SV function it takes no parameters, but rather looks at RTL design

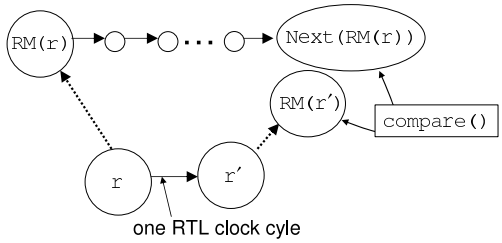


Fig. 3. Commutative diagram showing how our refinement checkers work.

signals internally via absolute signal names. Rule selection determines which sequence of rules to fire corresponding to each RTL clock cycle. When the Murphi rules are rulesets, rule selection must also determine the actual parameter values to use. In SV, rule selection is done by a function `Next(MURPHI_STATE ms)` that contains sequential code subjected to the restrictions that 1) all return statements in `Next()` return `ms`, and 2) all assignments in `Next()` having `ms` on the LHS have RHS of the form $R\_\mathtt{sv}(\mathtt{ms}, \ldots)$ (where $\ldots$ are the ruleset actual parameters). Since $R\_\mathtt{sv}$ errors out if invoked on `ms` for which R's guard is false, our restrictions imply that `Next(ms)` returns a `MURPHI_STATE` formed by firing *some* rule sequence from `ms`.

Fig. 3 shows how the checker operates using a commutative diagram. The current augmented RTL state `r` is first mapped through `RM` and then through the rule selection function `Next`. Then an error is raised if either during computation of `Next(RM(r))`, some $R\_\mathtt{sv}$ function is invoked when the guard is false, or if $\neg(\mathtt{Next}(\mathtt{RM}(\mathtt{r})) == \mathtt{RM}(\mathtt{r}'))$, where $\mathtt{r}'$ is the augmented RTL state in the next clock. A simply inductive argument shows that if the checker never throws an error, then there exists a Murphi behavior that the RTL simulation behaviorally refines.

Example: Toy Cache Controller. We now demonstrate these ideas through a toy example cache controller (CC). Murphi code for CC (abridged somewhat) is in Fig. 1. The state variables are:[3]

- `CacheArray` is an array of `CacheEntry`, which has fields State which can be `Invalid`, `Clean`, or `Dirty`, and a field each for the address and data.
- `Cpu2Cache` is a message being sent from the CPU to the Cache. To save space we omit the typedef.
- `Cache2Mem` is a message being sent from the Cache to main memory. To save space we omit the typedef.

Two rulesets are shown in Fig. 1. Both are parameterized by `i`, which is an index into `CacheArray`. `RecvStore` fires when a store command from the CPU is processed. The `Evict` ruleset is very non-deterministic in the sense that any valid line `i` can be evicted at any time. This is typical of non-deterministic HLMs; any RTL implementation will use some possibly complex eviction policy, but this complexity is abstracted away with non-determinism in the HLM. An RTL implementation is shown schematically in Fig. 4(a). This

[3]We note that a real design would have messages going from the cache to CPU and from memory to the cache.

implementation adds several details: deterministic eviction logic, two stages of pipelining, physically separate data and state/address arrays that get updated in different cycles.

Fig. 4(b) takes us through five consecutive RTL states with back-to-back store requests arriving on the `Cpu2Cache` interface, the first one requiring an eviction. Symbols A_i and D_i are used to denote addresses and data values, respectively. In cycle 1 the first store request (storing data D_0 to address A_0) arrives. Fig. 4(b) also shows the initial contents of the cache, which has A_1 dirty with data D_1, and A_2 clean with data D_2. In cycle 2 the first store is staged in the first pipestage, and the second store with address A_2 and data D'_2 arrives. Cycle 3 has the first store causing the eviction of A_1 from the cache, and the writeback is staged in pipestage 2. Also A_0 and `Dirty` are written into the evicted line. The corresponding data D_0 is written into the data array in cycle 4 and A_2 is made dirty. Finally in cycle 5 the new data D'_2 is written into the cache for A_2. Clearly three rules have fired in this example flow, i.e. two instances of `RecvStore`s and one of `Evict`. However, unlike the HLM, they are *non-atomic* (take multiple clocks to complete), and exhibit *true concurrency* (several rules can execute at the same time).

Because different data structures are updated at different cycles (i.e. non-atomicity), RMs must generally sample RTL signals at different temporal offsets from the current cycle. Ultimately this is done using history variables only. For CC, the RM must account for the fact that a store, which happens atomically in Murphi, takes three cycles in the RTL. Hence the map samples the incoming `Cpu2Cache` message, the state/address array, and the data array in consecutive clock cycles. Similarly the outgoing `Cache2Cpu` message is sampled in the same cycle that the data array is. The resulting RM is shown in Fig. 5, in which $signal@k$ is the value of $signal$, temporally offset by k cycles. In practice, we shift the RM so that all offsets are non-positive and thus implementable with history variables.

Rule selection deals with resolving true concurrency in the RTL. As discussed above, rule selection is encapsulated in a user-written SV function `Next` , which takes the current Murphi model state `ms`, and applies some sequence of rules to `ms`. This function for CC is shown in Fig. 5. Here we see that if there was a valid message 2 cycles earlier on the `Cpu2Cache` interface, then we calculate the target cache entry `i`. If an eviction is needed then we apply the appropriate `Evict` rule.[4] Finally we do `RecvStore`. In a realistic example there would be additional cases, e.g. for LOAD commands. Note that Fig. 5 results in either 0, 1, or 2 rules being fired per RTL clock. Fig. 4(c) shows the Murphi behavior corresponding to the RTL behavior of Fig. 4(b) and how they are connected by the RM.

III. Related Work

The most closely related work is that of Tasiran et al. [15]. Here the HLM was written in TLA+ and during simulation pairs of consecutive HLM states are passed off to the TLC

[4]The functions `get_target_cache_index()` and `eviction_needed()`, not shown, are user-written based on knowledge of RTL details.

```
function MURPHI_STATE Refinement_Map();
  MURPHI_STATE ms;
  for (int i = 0; i < CACHE_LINES; i++) begin
     ms.CacheArray[i].State = RTL.AddrArray[i].State@0;
     ms.CacheArray[i].Addr  = RTL.AddrArray[i].Addr@0;
     ms.CacheArray[i].Data  = RTL.DataArray[i]@+1;     end;
  ms.Cpu2Cache = RTL.Cpu2Cache@-1;
  ms.Cache2Cpu = RTL.Cache2Cpu@+1;
  return(ms); end;

function MURPHI_STATE Next(MURPHI_STATE ms);
  if (RTL.Cpu2Cache.Valid@-2) begin
     i = get_target_cache_index();
     if (eviction_needed()) ms = Evict_sv(ms,i);
     if (RTL.Cpu2Cache.Opcode@-2 = STORE)
        ms = RecvStore_sv(ms,i); end;
  return(ms); end;
```

Fig. 5. Cache Controller Refinement Map and Next function (rule selection)

model checker which checks if they are a valid HLM transition. Though not explicitly discussed in the paper, Tasiran et al. also allow multiple HLM steps per RTL clock; we feel that this is an important feature that deserves highlighting. Since their RM was expressed using 8K lines of C++ code, it is unlikely that one could use formal tools to reason about it, i.e. to prove that refinement holds. Contrarily, since our RM is expressed as synthesizable SV code, our approach is much more amenable to formal reasoning using standard tools.

Chen et al. use a form of Murphi called *Hardware Murphi* to verify hardware protocols [6]. It involves combining a traditional Murphi model with another model that specifies signal and timing info that can be used to generate VHDL. Other work includes compiling LTL or other formal assertions into dynamic checkers (e.g. [14]). These assertions involve RTL signals so no RM is necessary.

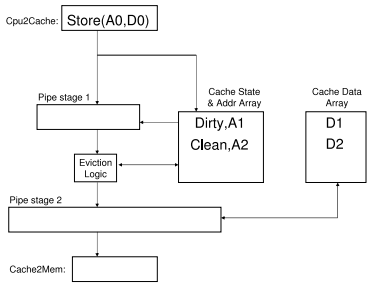
There have been a plethora of definitions for what it means for a lower-level system S_1 to implement a higher level system S_2. The importance of allowing S_2 to stutter was first promoted by Lamport [10]. Later Lamport and Abadi [2] showed that under certain assumptions, RMs can be shown to always exist, however their formalism does not allow the HLM to take multiple steps for each implementation step. To our knowledge, there are only a few definitions that allow for this, and many are in the context of superscalar microprocessor verification. These typically verify the existence of $t \in \{0, \ldots, k\}$ such that t HLM steps matches an implementation step. The bound k is the instruction fetch width of the implementation. For more on refinement for microprocessor verification see e.g. [1], [12].[5]

Finally, our work advocates manually writing a RM, rather than using more automated techniques such as flushing [5]. Like other work on RMs for real industrial RTL designs [9], [15], we have found manually-written maps more appropriate.

IV. Case Study

Our case study was a hierarchical cache protocol; the offchip protocol was Intel's QPI [11], while on-chip coherence is managed by a proprietary protocol that we'll simply call *Level-1 protocol (L1P)*. Our work was divided into two main parts:

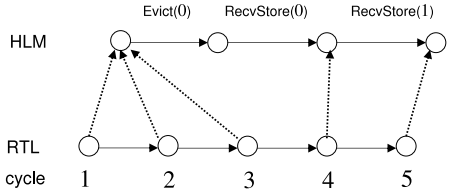
[5]We note, however, that verification of protocol implementations and that of microprocessor implementations are different beasts since the former operate in very restricted environments and have much more concurrency .



(a) RTL

cycle	1	2	3	4	5
Cpu2Cache	$Store(A_0, D_0)$	$Store(A_2, D_2')$			
Pipestage1		$Store(A_0, D_0)$	$Store(A_2, D_2')$		
Pipestage2			$Store(A_0, D_0)$ $WB(A_1, D_1)$	$Store(A_2, D_2')$	
State/Addr Array	$Dirty, A_1$ $Clean, A_2$	$Dirty, A_1$ $Clean, A_2$	$Dirty, A_0$ $Dirty, A_2$	$Dirty, A_0$ $Dirty, A_2$	$Dirty, A_0$ $Dirty, A_2$
Data Array	D_1 D_2	D_1 D_2	D_1 D_2	D_0 D_2	D_1 D_2'
Cache2Mem				$WB(A_1, D_1)$	

(b) Five consecutive RTL states



(c) Behavioral Refinement for behavior of Fig. 4(b).

Fig. 4. Cache Controller Example

first, building the QPI/L1P Murphi model and model checking it, then building refinement checkers for two key protocol components and using them during dynamic RTL simulation. These two checkers are now discussed

L1PM Refinement Checker. The central hub of L1P that handles processor core memory requests and also interfaces to QPI is a component called the *L1P master* (L1PM). We used our Murphi HLM and the methodology of this paper to create a refinement checker for L1PM. Developing the L1PM refinement checker took about 3 months of person effort, after the HLM and *mu2sv* had already been developed. The manual effort involved is developing and debugging the RM and rule selection. It was integrated into the validation test environment and run alongside the standard checkers and assertions for dynamic validation. After only one month, 8 RTL bugs had been found using the L1PM refinement checker.

QPI Home Agent Refinement Checker. The QPI Home Agent (HA) is the part of the memory controller that receives requests, sends snoops, and generally manages coherency [11]. Though at the time of writing this paper the QPI HA refinement checker was not yet deployed to the validation team, it is worth mentioning since it is considerably more complex than the L1PM checker. The checker involves seven different Murphi rulesets . Rule selection can yield up to 8 rule instances firing per clock cycle. The RM involves signals from five different pipestages and uses a window three clocks wide for temporal offset sampling. The HA refinement checker is still under development, but it passes a test suite of around 200 tests designed to exercise basic functionality.

V. Conclusions and Future Work

Though we believe refinement checking is an exciting approach to dynamic validation, ultimately we would like to *formally prove* that the refinement holds. Clearly, formal proof of behavioral refinement can be achieved by proving that the refinement checker never raises an error. However, we note that since all "ingredients" needed for refinement checking are required also for formal proof (except the test stimuli), and furthermore refinement checker development is a far less capricious activity, we advocate *starting* with checking and pushing towards formal proof only once the checker is in place.

References

[1] M. D. Aagaard, B. Cook, N. A. Day, and R. B. Jones. A framework for superscalar microprocessor correctness statements. *Software Tools for Technology Transfer*, 4(3):298–312, 2003.

[2] M. Abadi and L. Lamport. On the existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[3] M. Azimi, C.-T. Chou, A. Kumar, V. W. Lee, P. K. Mannava, and S. Park. Experience with applying formal methods to protocol specification and system architecture. *J. Formal Methods in System Design*, 22(2), 2003.

[4] R. Beers. Pre-RTL formal verification: an intel experience. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 806–811, 2008.

[5] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *6th Intl. Conf. on Computer-Aided Verification (CAV)*, pages 68–80, 1994.

[6] X. Chen, Y. Yang, M. Delisi, G. Gopalakrishnan, and C.-T. Chou. Hierarchical cache coherence protocol verification one level at a time through assume guarantee. In *High Level Design Validation and Test Workshop (HLDVT)*, 2007.

[7] N. Dave, M. C. Ng, and Arvind. Automatic Synthesis of Cache-Coherence Protocol Processors Using Bluespec. *Proceedings of Formal Methods and Models for Codesign (MEMOCODE'05)*, July 2005.

[8] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE Intl. Conf. on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[9] R. Kaivola. Formal verification of pentium ® 4 components with symbolic simulation and inductive invariants. In *Intl. Conf. on Computer Aided Verification (CAV)*, pages 170–184, 2005.

[10] L. Lamport. What good is temporal logic? *Information Processing*, pages 657–668, 1983.

[11] R. A. Maddox, G. Singh, and R. J. Safranek. The intel quickpath interconnect architecture. *Dr. Dobb's Journal*, May 19 2009.

[12] P. Manolios and S. K. Srinivasan. Automatic verification of safety and liveness for pipelined machines using WEB refinement. *ACM Trans. on Design Automation of Electronic Systems*.

[13] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1–3):279–309, 2000.

[14] K. Ng, A. J. Hu, and J. Yang. Generating monitor circuits for simulation-friendly gste assertion graphs. In *ICCD '04: Proceedings of the IEEE International Conference on Computer Design*, pages 409–416, 2004.

[15] S. Tasiran, Y. Yu, and B. Batson. Linking simulation with formal verification at a higher level. *Design and Test of Computers, IEEE*, 21(6):472–482, 2004.