Form Methods Syst Des (2017) 51:395–418
DOI 10.1007/s10703-017-0299-2

A methodology to take credit for high-level verification during RTL verification

Frederic Doucet[1] · **Robert Kurshan**[2]

Published online: 19 September 2017

Abstract High-level verification and synthesis of SystemC models has become increasingly popular as a means to reduce the high RTL verification cost of today's complex designs. However, the saving derived from performing verification at a higher level of abstraction is largely negated if the RTL then must be completely reverified. We demonstrate how global (system-level) properties may be verified at a behavioral level in a manner that reduces the required RTL verification. Our methodology entails using high-level control models together with semantic stubs for control and data-path refinements. The consequence is that cover goals met during high-level verification are then "virtually" met (in a semantically sound fashion) for RTL verification, and need not be re-established in the RTL. Moreover, it can be significantly more efficient (in terms of required verification cycles) to meet these cover goals at the higher level. This can lead both to less costly verification and earlier debug, providing a better structured, faster and more reliable path to implementation than is possible through conventional RTL verification.

Keywords SystemC · High-level synthesis · Micro-architecture · High-level verification · Coverage · Abstraction · Refinement

1 Introduction

The last decade has brought verification to the brink, with designs that are too complex to be adequately verified and even stories of commercial chip development cancelled because the design was ultimately deemed "too complex to verify". This has energized the electronic design automation (EDA) industry to seek qualitatively more powerful verification methods,

✉ Robert Kurshan
rkurshan@gmail.com

Frederic Doucet
fdoucet@qti.qualcomm.com

1 Qualcomm Technologies, Inc., 1700 Technology Dr, San Jose, CA, USA

2 New York, NY, USA

and this in turn has led inevitably to abstraction. In the last decade, High-Level Synthesis (HLS) tools have been successfully used in industry for multiple large designs at most of the large semiconductor companies. These tools operate at a higher level than traditional register-transfer level (RTL) synthesis tools. The input is a high-level model (HLM) commonly consisting of C++ functions describing untimed pipelined data-paths (or SystemC modules with a mix of control and untimed data-path functions). This HLM then gets synthesized into Verilog RTL. The main differences with traditional RTL logic synthesis are as follows:

1. the input C++ or SystemC code is abstracted from low-level microarchitecture details for resource sharing, multiplexing, finite state machines, and
2. the HLS tool characterizes data-path resources to decide how many pipeline stages are needed to implement an untimed function, and packs the logic in those cycles, automatically shares large arithmetic resources, and creates the sharing logic, finite-state machine (FSM) and all control bindings.

With the input code being free from such micro-architecture details, it can be much closer to the algorithm code, yielding multiple productivity benefits:

– faster time to RTL,
– lower bug rates due to verification on the input code, and
– earlier elimination of algorithmic bugs than if left to the RTL, and
– fast turnaround for large design changes.

Industrial HLS tools have now matured to the point that, when driven by a talented hardware engineer, the quality-of-results (QOR) of the generated RTL (in terms of speed and area, and potentially even other parameters such as power) will match or exceed what would be expected from a traditional RTL implementation. The potential to improve QOR while simplifying and thus accelerating development are principal selling points of HLS.

Using the SystemC language for creating a HLM of hardware renders the design easier to understand. Since it is smaller, it simulates faster. In practice, the higher-level of abstraction is inherent in the choice of the language itself: using a general-purpose software language is informally "more abstract" than using Verilog RTL simply because there are fewer details!

While the selling points of HLS are very attractive and have been demonstrated in many industrial projects, the deployment of the technology is not widespread and has proceeded more slowly than did RTL synthesis. One reason for this is that there is no clearly defined semantic connection between the HLM and the synthesized RTL, and thus no sound high-level verification (HLV) flow to pair with the HLS flow.

Although the HLM facilitates high-level verification, it fails to utilize positive high-level verification results to relax the burden of RTL verification. Without a semantic relationship between the HLM and the RTL synthesized from it, all properties verified in the HLM must be reverified in the RTL. Thus, utilizing HLS requires high-level properties to be verified twice.

Nonetheless, HLS can keep all its advantages while eliminating the disadvantage of increasing the verification burden, if a semantic connection between the HLM and RTL is established, rendering properties verified in the HLM automatically correct in the RTL as well.

While there is considerable production usage of HLS methodologies where it may commonly be understood that the HLM is an "abstraction" of the RTL because the HLM is simpler, easier to understand and meant to represent the RTL, high-level abstraction has not been well-defined in the HLS flow.

The absence of sound abstraction in the HLS flow creates two significant issues with the methodology.

First, one may run many tests on the abstract design, but with the HLS flow, as already explained, there is no guarantee that once the tests pass on the abstract design, the same set of tests will pass in the generated RTL. Although many if not most re-run tests that passed in the HLM may also pass in the RTL, they all must be re-run on the RTL, and the few that passed in the HLM but failed in the RTL could require time-consuming fixes. There are additional required RTL checks that the increased latencies inserted by the HLS tool do not break the design functionality. (Of course, no matter the methodology, there always will be a requirement to run low-level checks for behaviors unrepresented in the HLM, so some checks in the RTL are inevitable.)

Although functional debug at a higher level of abstraction can accelerate design development by eliminating functional bugs earlier in the design flow, doing verification at each level of abstraction nonetheless can increase verification costs beyond the cost of conventional RTL verification alone. This is the case even after accounting for the verification acceleration advantage of earlier high-level debug. The added cost of repeated verification at multiple levels of abstraction (even just two), has created a barrier for the utilization of HLS.

The second issue with HLV is that "coverage closure" is not well-defined in the HLS flow. Coverage closure refers metrics that establish the adequacy of a suite of simulation tests. Although these metrics may be defined for the RTL, there has been no automatic way to lift these metrics to the HLM. This is especially problematic with regard to technology transfer of HLS to RTL design teams accustomed to an RTL design methodology.

A semantic connection between the HLM and the RTL could be used to automatically "lift" coverage closure metrics from the RTL to the HLM. Many randomized RTL tests could abstract to the same HLM test and thus need not be run, opening the HLM testing budget to a broader variety of tests than could be run on the RTL.

As mentioned above, in contrast with formal behavioral verification, there is a fundamental problem with the way the HLM is written in SystemC and then synthesized using HLS: the two models are not semantically consistent, i.e., the HLM is not necessarily a conservative abstraction [1] of the RTL. This can work against the goals of HLS that are meant to provide a more transparent representation of the RTL. While the current HLM structure has clear benefits, as just explained, it unfortunately thus deprives the designer of the possibility to use the HLM to accelerate design verification adequately, through HLM verification.

In this paper we show how to use high-level models, SystemC and micro-architecture synthesis directives, to implement a hierarchical design and verification methodology wherein high-level verification coverage can be inherited to the RTL coverage. This allows one to take credit for HLV (both simulation-based and formal), thereby rendering superfluous any RTL verification of system-level properties already verified in the HLM. Block-level verification of data-path functions still is required, but since the data-path verification can be applied locally to the data-path functions, this flow provides a verification methodology that can scale with increasing design size.

Specifically, with an HLM that is a sound abstraction of the RTL design, compositional techniques that are routine with formal verification may be applied to the simulation-based flow as well. These techniques ensure that any property valid in the HLM is inherited by the RTL. With formal verification, "valid" means "formally verified". When using simulation-based testing, the definition of "valid" can be weakened to mean "tested with adequate coverage".

Since the HLM is simpler than the RTL design, any level of coverage obtained in the HLM is in effect multiplied by the ratio of complexity between the RTL design and the HLM. Thus, even once the RTL has been fully synthesized, if additional simulation cycles are required to test system-level properties, it would be more efficient to perform those tests in the HLM than in the RTL. (Naturally, before one is confident of the high-level verification methodology, verification would be redundantly repeated in the RTL. Eventually, with increased confidence, such repeat verification will naturally be eliminated as test resources are switched to more urgent priorities.) All this is possible only if the RTL is semantically consistent with the HLM, in other words, if the HLM is a sound abstraction of the RTL design (in a formal sense).

Our abstraction methodology is based on using “semantic stubs” as place-holders for lower-level data-path elements, during the coding of the HLM. These stubs are similar to the place-holders for code not yet written that are routinely used in programming, but are different to the extent that some representation of “inputs” and “outputs” (reads and writes) are required for these stubs. These inputs and outputs are abstractions of the RTL data-flow. For example, consider an ALU that returns a value from data inputs. The semantic stub may just receive a token (input abstraction) that triggers an abstract computation, the result of which is the output token *done*, standing as an abstraction of the returned computation. With formal verification, the stub would use nondeterminism to abstract the latency between the receipt of the input token and the generation of the output token. When simulation is used, the nondeterminism is replaced by a random variable that determines when the computation is complete. (Note that in either case, the stub requires a bit of state to remember if a computation is “in progress”.) Likewise, a stub can abstract control features through the use of nondeterministic or randomly chosen output control tokens, which can be constrained to align with required behaviors.

Validation of the computation is deferred to the RTL. However, many aspects of the control flow can be verified in the HLM without invoking the actual computation. In this example, the stub provides for verification under all possible latencies. When such generalities are not required or not allowed, bounds on latency can be defined in terms of events: the latency is constrained so as not to exceed the time to some other event that expects the returned result from the ALU. (Constraints are compatible both with formal verification and constrained random simulation.) Constraints used to verify the HLM become proof obligations (or RTL requirements) to be verified in the RTL. This methodology thus also nicely partitions system level behaviors and RTL behaviors. The system level behaviors are verified subject to assumptions about the RTL behavior and then these RTL assumptions are verified in the RTL.

Such partitioning can further accelerate verification when the assumptions on the RTL can be verified in isolated RTL blocks, thus speeding and simplifying their verification and (in the case of simulation) improving coverage as a result of simulating blocks instead of the entire design.

While this is at considerable variance with conventional simulation that runs the entire design altogether, it is expected that with time, current simulation methodology will give way to the more efficient and more reliable block-based simulation described here. Even if the conventional system-wide simulation methodology is retained, efficiencies can be realized by limiting the simulation targets to the RTL behaviors assumed during the system-level verification of the HLM.

The experience of transferring formal verification technology to industry [2] suggests that transferring the methodology proposed here will be no easier. However, within two decades, that transfer succeeded through the persistent efforts of many researchers. It undoubtedly would not have succeeded, however, without an increasingly dire need in the industries to overcome what was quickly becoming an unsupportable debug problem. Designs had become too complex to implement correctly within a commercially supportable timeframe, without a new means to catch functional design errors earlier in the design cycle than was then possible. Even so, to incorporate formal verification into the design/development flow all at once would have been prohibitively disruptive. The key was to devise means to adopt formal verification incrementally, first as a parallel path, then as a simple largely automated but correspondingly weak process that finally gained power as the user base became more sophisticated. This allowed the EDA tool vendors to profitably market ever more sophisticated formal verification tools. There is every reason to suppose that the current proposal could follow the same path.

Writing semantic stubs can be significantly simplified through the creation of stub libraries for commonly used elements. Such stubs would be parameterized, and various configurations could be pre-verified, further simplifying their application. Helmut Veith has made many important contributions to the verification of parameterized systems [3–7], which could be used in support of such a program. This underscores the great loss to our profession of Helmut, in the prime of his creativity and contributions, to whom this volume is dedicated. Helmut was also a colleague highly admired for his inspirations and humanity, and is greatly missed.

This paper is organized as follows: in the next section, we review the semantics of SystemC for synthesis, and show how HLS can render the input and output designs semantically incompatible, introducing coverage holes. We discuss how abstraction based on higher-level modeling with semantic stubs can be used with synthesis directives to produce a sound refinement. In the third section we present our methodology of encoding a global control model that is step-wise refined in a semantically sound manner through the use of semantic stubs, so that the resulting RTL design is semantically consistent with the HLM. We show how coverage can be closed in the context of HLS through this methodology. Section 4 presents a small illustrative example of our methodology, and Sect. 5 discusses some related work.

2 Semantics of high-level models for synthesis

In this section we define the semantics of synthesizable SystemC in the HLS context, and discuss abstraction and refinement, setting the stage for a hierarchical compositional methodology presented in the next section.

2.1 Overview of high-level synthesis

High-level synthesis tools are used to synthesize SystemC into Verilog RTL. Figure 1 shows the three typical inputs to such tools:

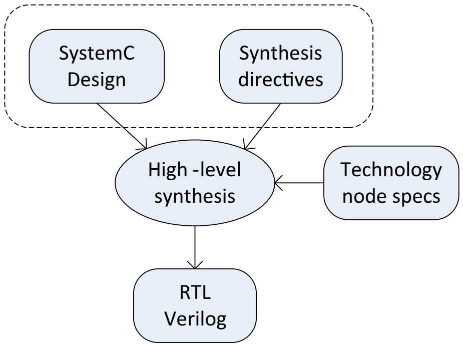


Fig. 1 Inputs to high-level synthesis tools

1. SystemC code for a module to be synthesized into Verilog RTL,
2. Description of the technology node, clock frequency etc., and
3. Synthesis directive for selecting micro-architecture implementations for the SystemC syntactic structures.

The core engine in a typical HLS tool is the scheduler, which allocates resources (adders, multipliers, etc) and maps arithmetic and logical operations (add, multiply etc) onto them. The goals of the scheduler are to:

1. schedule the operations within given constraints: minimize the number of resources for the given latency budgets;
2. generate RTL that will be synthesized by a logic synthesis tool, with predictable timing closure and power efficiency.

For this, the scheduler will characterize all resources in the design for their timing and area with the selected technology node. Generally, it will use this information to build the RTL by filling the clock cycles (for a given frequency specification) with just enough logic so that the generated RTL will close timing in logic synthesis. Typically, design latency will be increased by the scheduler, as it builds the computation pipelines, to have positive slack or minimize the design area. The designer provides synthesis directives to guide the tool engines to specify how to implement specific SystemC structures, as well as constraints for resource creation and latency budgets.

2.2 Interpretation of synthesizable SystemC syntax

The SystemC and Verilog languages are used to describe hardware modules containing concurrent processes, communicating through signals. Both these languages have the same semantic foundation. We restrict the HLM to use only a synthesizable subset of SystemC that directly and unambiguously translates into hardware structures as RTL designs. (We omit the semantic translation rules of SystemC.) This effectively simplifies the semantics of SystemC to the classical synchronous semantics, enabling efficient analysis with existing commercial Verilog simulators and model checkers.

We require all the external interfaces of components to be at the signal-level where each transition is from/to a program counter location that denotes either the initial node of the program graph, a node which corresponds to a wait statement, or the exit node of the graph. Generally, a process will have a sequence of statements between two wait statements; these must be written so as to be transitively collapsable into a single "macro" transition into the transition system. The intermediate micro-transitions between the other SystemC statements likewise must be constructed to be collapsable into these macro-transitions. Additionally, any combinational processes leading to state variable assignments (i.e., to flops) are similarly collapsed, so that we can reason on a single transition to successive steps.

2.3 Semantic framework for high-level and RTL models

Next we describe the basic framework on which the module descriptions are characterized, and how to relate their behaviors. A hardware design description in SystemC or Verilog is the definition of a module, with input and output variables and processes. Behavior of a process is formalized by a transition system.

Definition 1 (*Transition system*) Let V be a set of Boolean input and output variables $V = (V_I, V_O)$ and $\Sigma : V \mapsto \{0, 1\}$ is the state assignment function. A transition system is a tuple (Σ, σ_0, T) where $T \subseteq (\Sigma \times \Sigma)$ are the state transitions and $\sigma_0 \in \Sigma$ is the initial state.

The behavior of a process is characterized by observing, at each clock cycle, the sequence of values at its inputs and outputs. Σ is the alphabet of M, and the observations of state assignments are sequences of words $\sigma_0\sigma_1\sigma_2\sigma_3\ldots$ Such a sequence we call a trace of the process.

Definition 2 (*Trace*) Let $M = (\Sigma, \sigma_0, T)$ be a transition system for a process P. A *trace* of P is is a sequence of state assignments $\sigma \in \Sigma$ synchronous to a clock: $\sigma_0\sigma_1\sigma_2\sigma_3\ldots$ where for each pair of successive states in the trace, $(\sigma_i, \sigma_{i+1}) \in T$.

A state σ in a trace includes both the input and output values, which will determine what the next state will be. Input values are controlled by the environment, and output variable are controlled by the design module. Output values can also include the values of internal variables, which are not externally visible.

Definition 3 (*Behavior*) Let $M = (\Sigma, \sigma_0, T)$ be a transition system for a process P. The *behavior* of P is characterized by the set of all possible traces for P, denoted *traces*(P).

Since V_I is controlled by the environment, the behavior of P is its reactions: the assignments of all the variables in V_O, for all possible input values.

2.4 Example: behavior of SystemC module

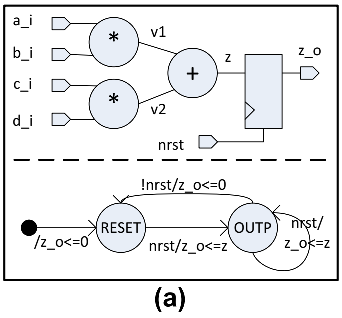
In this subsection, we illustrate the semantics of a SystemC description with an example of the small design listed in Listing 1. First, there is a `compute()` data-path function multiplying its inputs `a` and `b`, and inputs `c` and `d`, adding and returning the result. The data-path function is wrapped into a SystemC module where a synchronous process reads the input values from the input ports and calls the compute function, and writes the result on the output signal port. There is only one `wait()` statement, at the bottom of the while-loop, meaning all arithmetic operations occur in a single clock cycle. Let us designate this version of the design by M_1.

Listing 1 Example code for SystemC module wrapping a data-path function

```
// datatpath function
int compute(int a, int b, int c, int d, int d) {
  int v1 = a*b;
  int v2 = c*d;
  int z = v1+v2;
  return z;
}

// SystemC module
SC_MODULE(DUT) {
  sc_in <bool> clk; // input clock
  sc_in <bool> nrst; // input reset signal
  // input ports, 32 bits signed
  sc_in <int> a_i;
  sc_in <int> b_i;
  sc_in <int> c_i;
  sc_in <int> d_i;
  // output port
  sc_out<int> z_o;

  SC_CTOR(DUT) { // declare process sensitive to posedge clk
    SC_CTHREAD(process, clk.pos());
    reset_signal_is(nrst, false);
  }
```



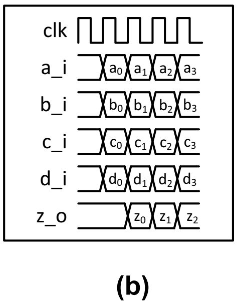


Fig. 2 Design version M_1: cycle-accurate SystemC **a** interpreted circuit structure, **b** I/O trace behavior

```
void process() {
  z_o = 0; // Reset section, initialize all output flops to 0
  RESET: wait(); // end of reset section
  MAIN_LOOP: while (true) {
    // Read inputs and call data−path function.
    int z = compute(a_i.read(), b_i.read(),
                        c_i.read(), d_i.read());
    // Empty statement for latency increase.
    INCRLAT: ;
    z_o.write(z); // Write outputs to D pin of flop.
    OUTP: wait(); // Wait for next clock cycle (flops D−>Q)
  }
 }
};
```

The direct interpretation of the configuration version M_1 (or a high-level synthesis in cycle accurate mode) is shown in Fig. 2a, where we see the data-path implementing the computation in the upper part, and the state machine in the lower part. The data-path is very simple, with two multipliers, one adder, and one output flop. The state machine has only two states:

1. the reset state, where the output flop z_o is initialized to zero, and
2. the output state, where the result z of the computation is assigned to the output flop (the inputs are always flowing through the multipliers and the adder, computing the result in a single cycle).

The depiction of the state machine shows labeled states and edges. The encoding of a state in Σ is for the state label with the assignments of the controller and observed variable on an incoming edge (e.g. OUTP/nrst/z_o = z). Figure 2b shows the observable trace for the process, where one can observe that the computation has a latency of a single cycle (each output is always available in the cycle following each respective input value).

2.5 Semantic changes through micro-architecture directive

Next, we illustrate the semantic differences of input and output for two of the most important micro-architecture directives used with HLS: latency constraints and pipelining.

In the previous sub-section, we described the synthesis of the SystemC description with a architecture directive for cycle-accuracy, yielding RTL cycle-accurate to the SystemC. While this is an important use case, most of the time, the HLS input will usually include micro-architecture directives specifying latency and throughput constraints, or how to implement specific SystemC constructs (such as unrolling a loop), changing the behavior of the input model. This is typically used to synthesize data-path functions in a pipeline: the function itself does not fit within a single cycle and needs to be spread over multiple pipeline stages. In that sense, the behavior of generated RTL is defined by both the input SystemC code as well as the micro-architecture directives fed to the HLS tool.

We demonstrate how micro-architecture directives change the I/O behavior of the synthesized model with the example listed in of Listing 1. For this, let us also provide as input to the HLS a technology library where multipliers taking 4ns and adders 1.2ns, and specify clock period of 5 ns.

2.5.1 Synthesis directive: latency constraints

Latency constraints are used to permit the scheduler to increase the latency of the design and move the operations to the added states. Adding cycles can help with timing or to share resources for area reduction. Let us denote by M_2 this version of the design example, where the input code in Listing 1 is paired with the synthesis constraint to increase its latency the at INCRLAT.

The synthesis results in the hardware structures are illustrated in Fig. 3a, where one can see that the scheduler added two states, moved the multiplier to the second and the adder to the third state. With this structure, the synthesized design can meet timing and use only one multiplier to perform two multiplications. It also allocated several registers and sharing multiplexers around the multiplier to share it for two operations. One can also observe in the state machine when the registers are written and multiplexer paths selected. Figure 3b shows the cycle behavior for the generated micro-architecture. The latency is now three cycles, and the throughput is one input being read every three cycles.

Strictly comparing the behavior of synthesized model M_2 with the original SystemC model M_1, one can observe that, while the variables are the same, the observable behaviors are completely different because of the change in I/O timing. These are completely different sets of traces: $traces(M_2) \cap traces(M_1) = \emptyset$ because even if the input is held constant for three cycles, the latency to output will be different.

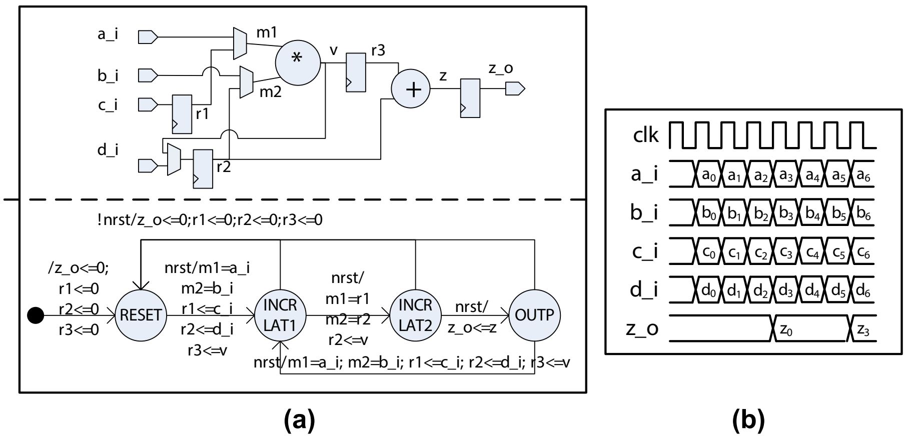
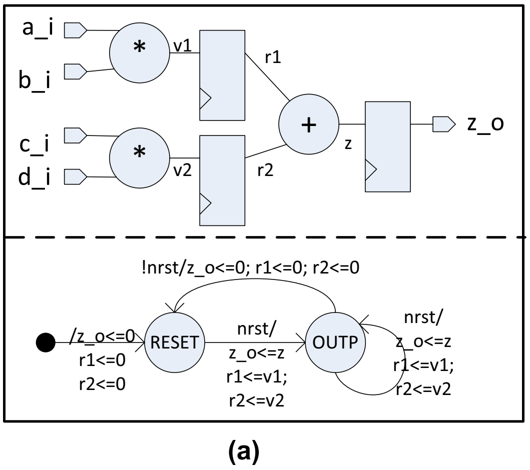


Fig. 3 Design version M_1: synthesis directive to increase latency: **a** generated structures, **b** I/O behavior



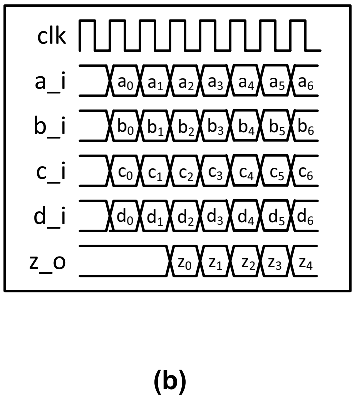


Fig. 4 Design version M_3: synthesis directive for loop pipelining: **a** generated structures, **b** I/O behavior

2.5.2 Synthesis directive: loop pipelining

The second micro-architecture directive we consider is one to transform a loop in the SystemC code into a hardware pipeline. This directive will instruct the HLS tool to spread the compute function onto a pipeline structure. The depth of the pipeline is the number of stages necessary to get positive slack. Keeping with the example in Listing 1, we instruct the HLS tool to create a basic pipeline by adding pipeline stages at the INCRLAT statement in the SystemC code.

The synthesis will result in the structures showed in Fig. 4a, where the data-path has two multipliers in the first stage, and the adder is in the second stage. The state machine shows only two states: reset and output. In the output state, the pipeline is active and with all registers enabled to store the values through the pipe. Figure 4b shows the behavior of the generated RTL, with latency of two cycles, but throughput of one cycle (one value out every cycle once the pipeline is full). Here again, while the set of variables are the same, the observable sequences are very different beecause of the initial latency: the sets of traces of the synthesized model has no relation to the set of traces of the input model: $traces(M_3) \cap traces(M_1) = \emptyset$.

2.6 HLS verification in practice

As we have seen in the previous two subsections, the output of HLS can have different I/O behavior than the input SystemC. This is typical to HLS usage. While semantically this quite apparently is a big issue, it has not stopped its use in industrial production. It is a fact of current practice that HLS is being used and designs are being verified and taped-out. However, not surprisingly, this semantic disconnect commonly introduces serious bugs that can require extensive effort to correct.

For data-flow designs, the main use case for HLS, this can be done with a procedure illustrated in Fig. 5: simulate the SystemC version of the design under test (DUT), synthesize it, and then re-run the same tests (and perhaps more) on the RTL version of the DUT. The key to this procedure is that the calls to the data-path function must behave like a “transaction”: an untimed function that, given a specific input, will produce a specific expected output. Then, the sequence of input and output values can be checked during simulation: a driver

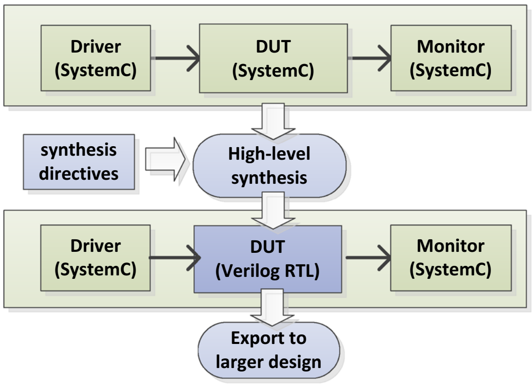


Fig. 5 Sandbox verification flow around HLS: driver and monitor use handshakes to account for I/O behavior changes introduced by the HLS

will feed the input values to the design at designated times, and a monitor will check for them as required. To handle the I/O behavior changes introduced by the HLS, the driver and monitor typically will use some kind of handshakes to denote the event when a value should be driven or a check should occur. It is a common design practice to make those robust to design throughput and latency changes using data-valid handshakes, to check only when it sees the output valid signal asserted. With such techniques, the events in the input model traces can be related to the events in the generated RTL traces. Typical procedure is for the driver to feed the DUT with a "golden" input trace thought to define correct behavior, and for the monitor to assert that the DUT output matches golden output traces (large designs will typically have a very large number of such trace files). If the design and test bench are robust to I/O behavior changes, this methodology is straightforward and can be efficiently implemented by design and design verification (DV) teams.

While re-running all tests on both models may be practical for block verification for data-flow style hardware accelerators (where the data-path functions are synthesized in a straightforward way), for larger submodules (and complete subsystems), it may become much more complicated to align input and output models. Also, often RTL design engineers will forgo handshakes due to the area/latency cost and complexity that these incur, and instead use timers in the system. Thus, while transaction-based verification works well for data-flow blocks in the design, for more control-oriented blocks in the design, the nature of the verification is different.

This effort of trace alignment and of repeating the verification on the RTL could be avoided by taking a little care during synthesis to maintain the soundness of the HLS by ensuring that the SystemC model is indeed a semantic abstraction of the synthesized design.

2.7 Abstraction in HLS flow

Industrial practitioners (hardware designers, DV engineers, EDA tool developers, etc.) often think the SystemC model is more abstract in an informal sense because it is usually contains fewer lines of codes, is easier to debug, faster to correct, etc. As the examples in the previous subsections suggest, the designer does not need to explicitly (or at all) specify:

- operation to resource mappings (with resource sharing and multiplexer creation and wiring)
- allocation, sharing, and mapping of registers for all internal control and data values,

– creation of a control finite state machine (FSM), with exact cycle mapping of control bits to/from the data-path in the required clock cycles, and
– creation and wiring of register and resource clock gating logic.

While relegating these to the synthesis tool clearly provides benefits (much less structural details for the designer to design), it does not articulate a precise and formal meaning for abstraction in the HLS context. By contrast, we argue to formally establish the meaning of abstraction in the HLS flow by using the standard automata-theoretic approach to formal verification such as described, for example, in [1].

2.7.1 Property verification

A property P defines a set of $\mathit{traces}(P) \subset \Sigma^{\omega}$ over the observable variables that is described using an ω-automaton. An ω-automaton is a sequence acceptor, defining a *language* $\mathcal{L}(P)$, where a trace is in the language if and only if a set of its states or transitions is visited infinitely often in a manner that conforms with the automaton *acceptance* condition. This type of automaton specifies safety and liveness properties: requirements for what must not happen, and, respectively, for what must happen repeated or eventually.

A design M is formalized as a transition system represented by an automaton whose traces comprise the set of possible executions of the design.

The verification of a property P over a design M is checking the language containment $\mathcal{L}(M) \subseteq \mathcal{L}(P)$; this can be performed through well-known decision procedures that have been implemented in commercial model checkers for at least the last two decades. For safety properties, this procedure is commonly effective for many large design blocks. Liveness properties typically require more computational resources.

2.7.2 Abstraction and refinement

We use the standard definition of refinement as one of trace or formal language containment: the abstract model has a behavior that is a superset of the traces of the refined model. We use the language of a property (or the more abstract design) to provide the basis for testing the soundness of a refinement (namely as automaton language containment).

Definition 4 (*Refinement*) Let M_1 and M_2 be two transition systems, and π a map of events in M_2 to events in M_1. Then M_2 is a *refinement* of M_1, denoted $M_2 \preceq M_1$, and equivalently, M_1 is an *abstraction* of M_2, if and only if $\mathit{traces}(\pi M_2) \subseteq \mathit{traces}(M_1)$.

Thus, any property that is valid for an abstraction M_1 remains valid for a refinement M_2, relative to the map π.

Typically, a system M is composed of n submodules, $M = M_1 || \ldots || M_n$. A global property P specifies a behavior governing the interaction of all components, requiring the global verification of $\mathcal{L}(M) \subseteq \mathcal{L}(P)$ to be performed. The more abstract the model, the easier it is computationaly to perform this check. A local property P_1 can be checked locally on a sub-component M_1 with $\mathcal{L}(M_1) \subseteq \mathcal{L}(P_1)$, and a M_1 can be refined into $M_1' \preceq M_1$ with more details (and more local verification $\mathcal{L}(M_1') \subseteq \mathcal{L}(P_1')$). For the refinement to be valid, and for P to hold on the last refined model $M' = M_1' || \ldots || M_n'$, the refinement relation $M_1' \preceq M_1$ must be checked for each sub-component refinement. An abstraction is *conservative* if it is an abstraction of an inferred refinement as per the above definition.

2.7.3 Controlled refinement to avoid expensive consistency checks

The goal of the methodology described in this paper is to provide a framework where the refinement is correct by construction, thereby avoiding expensive refinement checks. To that end, abstraction is introduced in a model using non-determinism or by datatype and selection abstraction. Specifically, we require the entire high-level control structure to be preserved through all refinements, including a special nondeterministic "pause" self-loop transition that finally resolves to a specific delay in the implementation.

The only exception to the "valid by construction" refinement is in the case of fairness constraints added to these "pause" self-loops, in order to be able to verify eventuality properties in the abstract level. Examples of such fairness constraints include an assumption that a computation eventually must terminate, or an assumption that some appropriate response eventually is sent. However, such fairness constraints are required only in the case that there are liveness properties whose verification is required in the high level model. An example of such a liveness property might be that some task is eventually completed.

If all the properties are safety properties–properties that stipulate that something "bad", like buffer overflow, can never happen, then there is no need for fairness constraints, and the verification process is completely "by construction": no computational verification is required.

Once the nondeterminism is resolved, any fairness constraints become proof obligations that must be verified on the lower level at which the nondeterminism is resolved. Again: these are the only not-by-construction verifications required.

There can be other properties as well that must be verified at lower levels, namely properties involving behaviors not defined at higher levels. In the design process, the high-level control structure is preserved: only nondeterministic transitions can be resolved in implementation (removing the need for the fairness constraint), and data-type and selection can be refined if it preserve the abstract alphabet through map π.

All other refinement needs to preserve the sequences (through the map π). If a given refinement requires breaking a module into submodules (introducing more concurrency), then the language containment must be checked to discharge the verification of any fairness constraint (that the design shall not stall, for example). But the goal is to avoid these kind of checks, and to capture the concurrency in the abstract model, proving all local properties there. If more concurrency needs to be added, then the designer needs either to return to the abstract model or perform the refinement check.

This "controlled refinement" rule is the foundation of our methodology. If HLS is performed in a manner that produces a formal refinement in this sense, any verification performed on the abstract model need not be repeated on the implementation. The examples in preceding subsections show that the traces of the SystemC input may not support any map to the corresponding variables of the traces of the generated RTL model. To be able to do this we thus require a formal alignment of transactions entailing:

1. auxiliary processes to model the driver and monitor behavior and auxiliary variables for aligning the traces, and
2. silent transitions in the models that may be transitively collapsed into traces that can be compared.

If not accomplished through a structured design methodology, such structures are quite tedious to create and align, and it is not something done in practice because the formal verification algorithms typically work well only on the control-oriented parts of the design and do not scale well on the data-path functions (which are usually removed by abstraction, but with increasing use of word-level formal verification, this may be changing.)

For the proceeding examples, with their I/O changes, and for the HLS flow in general, it is thus clear why the verification (formal or simulation) needs to be fully redone on the implementation model, with all tests redefined and re-run. To avoid this crippling cost, and associated design development delays, is the motivation for the methodology presented next.

3 Design and verification methodology

Now we propose a design and verification methodology built around a HLS flow that preserves the HLM as a semantic abstraction of the generated RTL design. The first step is to define a map π from the events of the generated RTL to the HLM. To accomplish this we refer both to a SystemC description of the HLM and the HLS micro-architecture synthesis directives. Our methodology incorporates two aspects:

1. A top–down design methodology for global control-oriented behaviors and the design's data-flow, and
2. A bottom–up coding methodology for data-path functions.

Verification is integrated into each methodology. For the first, one starts with a HLM model, verifies properties relative to it and then refines it to a lower level model. In practice, the top–down methodology establishes a hierarchy of models, rather than proceeding from a single HLM to the RTL design in a single step. Each of the intervening models is also a HLM, although it may be a refinement of a more abstract HLM.

At each level of the top–down hierarchy, properties are verified using assumptions, if necessary, that eventually get verified at lower levels of the hierarchy. The higher-level models typically are more control-oriented, with the consequence that in the systematic top–down approach, control gets implemented before data paths. This is consistent with modern design practices [8] for control-dominated designs, permitting the designer to get functional debug feedback as soon as the control-flow is coded, well before the completion of RTL coding of data-path functions.

It is well-known that this design style affords earlier debug, which can significantly accelerate design verification closure by highlighting functional bugs while the design is still fluid. Likewise, it comes early enough in the design development flow to permit inexpensive redesign.

The bottom–up methodology is focused on the correct implementation of micro-architectures and data-path functions that are inherent in the design. Since these have very specific implementations, it is important to start with appropriate coding for them, and then abstract the codings for use in the HLM verification. These abstractions have the form of the "stubs" mentioned in the introduction. These two methodologies "meet in the middle" to form a design hierarchy. Each component of the bottom–up methodology tends to be "local", in the sense that its behavior can be defined and verified without reference to the rest of the design. An ALU is an example.

3.1 Semantic stubs for controlled abstractions and refinements

In our methodology, we use semantic "stubs" as place holders for further refinements, as well as abstractions of low level structures. Soundness of refinement can be guaranteed by adhering to the following rules:

1. abstract model must either include the micro-architecture synthesis directives, to have same I/O timing behavior as the implementation, or else be left nondeterministic, with the timing to be resolved in the refinement;
2. symbolic values represent datatypes that are refined in implementation;
3. control input/output ports can be expanded with data ports;
4. case-statement branches can be expanded if they do not change the observable I/O behavior;
5. nondeterministic latencies can be refined into actual implementation latencies of the computation in data-path functions.

The stub rules provide the framework for a conservative abstraction and refinement by introducing non-determinism (implemented as random choice for simulation) to abstract both latency and low-level computations, when the precise latency or computation is not germane to the level of abstraction. Both uses of nondeterminism can be constrained with assumptions about their selections relative to contemporaneous events in the model.

For example, a nondeterministic latency can be constrained to end no sooner than the "ready" signal of a peer component, and no later than some other event. Likewise, a unit that may return a selection of error codes may be abstracted by a stub that returns only two codes, say, "error" and "no error", and this selection may be constrained to never return "error" in cases that that selection is incompatible with other model actions. All such constraints comprise proof obligations that need to be verified in the implementation.

A representation of the latency associated with a computation is fundamentally important, of course, and can not be set to span fewer clock cycles (or align with the completion of fewer events) than will be possible in the implementation, as doing so could mask concurrency bugs such as conflicts or race conditions. On the other hand, conservatively generalizing latency beyond what is actually possible in the implementation can help illuminate concurrency bugs that may not be present in the current design, but could manifest as the design evolves. Designers have found it very useful to have such *potential* bugs illuminated (although they may not be possible in the current implementation), as understanding their potential can lead to a more robust design. Moreover, understanding such concurrency "pitfalls" can lead to deeper insights into the design behavior that can be very useful as the design evolves.

3.2 Unifying top–down and bottom–up methodologies

In the top–down methodology, one begins with a hierarchy imposed by the given design specification. Design specification properties (defined by simulation monitors or formal verification assertions) are partitioned according to their respective extent of locality. More global (system) properties sit higher in the hierarchy. A system design hierarchy is induced from the specification hierarchy.

The granularity of the HLM is determined by the degree of locality of respective design specification properties. The more global properties give rise to more abstract HLM models that are utilized to verify those properties.

Figure 6 illustrates the global and local flows with an HLM example. We see two modules, the first one with a state machine receiving a start signal, with local values written into the memory on data-valid event (the transition and the directions of the variable indicate the observed and controlled variables; self-loops are omitted). The values read from input and written to the memories are abstracted out of this model. The state machine in module 1 will write values to the memories until the synthetic non-deterministic (or random) input `nd1` is asserted. Then, it will trigger module 2 with the `go` signal. Nondeterminism is easy to

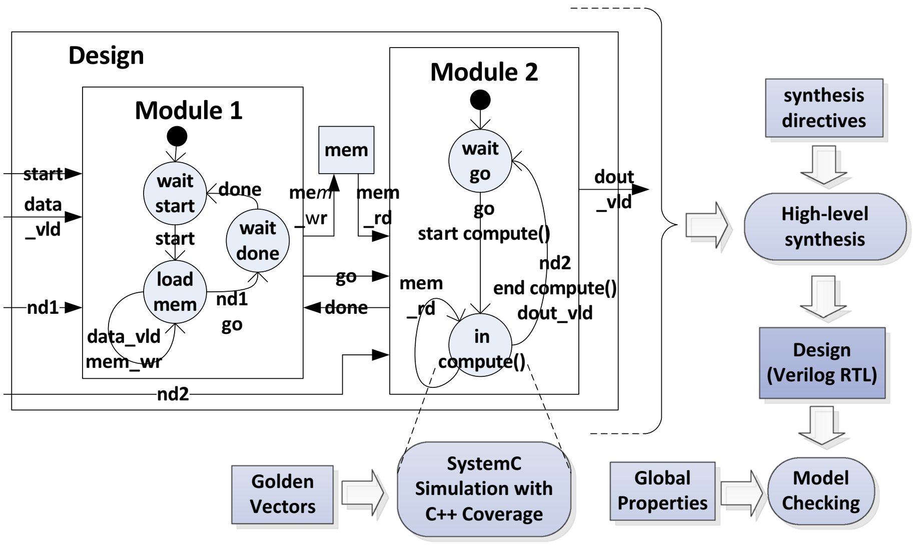


Fig. 6 Example of global top–down and bottom–up refinement flow

encode: it is implemented through the introduction of a synthetic input; for simulation, it is randomly assigned.

Upon reception of the `go` bit, the second module will call the `compute()` function. The `compute()` function will read values from the memories until the non-deterministic signal `nd2` is asserted. Then, `compute()` will end and return the output with the output data value asserted. Here, the latency of the `compute()` data-path function is abstracted by the nondeterministic signal `nd2`.

3.2.1 Global verification

For the global verification, we go through the high-level synthesis, where a directive of latency constraints is applied to the compute function, setting it either to a nondeterministic value (for higher-level verification), or to a specific number of cycles in the ultimate RTL design. The input model is synthesized into Verilog, and the global control verification is done with a Verilog model checker or simulator against a given set of properties. For simulation, this provides the coverage for all possible control paths covered in the HLM.

3.2.2 Local data-path function verification

The local verification is typically done by simulating the data-path function and using "golden" reference vectors for correctness. These are vectors that are believed to define correct design behavior. (Unsurprisingly, they are virtually never complete in this regard, but it's what's available to simulation-based verification.) This testing may be done with a transaction-level test, where we call the function with the input arguments, and verify that the outputs are as expected. We can use a C++ coverage tool to ensure that all the paths and conditional expression terms in the source code of the compute function are covered. This provides us the means to test and debug directly with C++ code and close the coverage at that lower, more local level of abstraction.

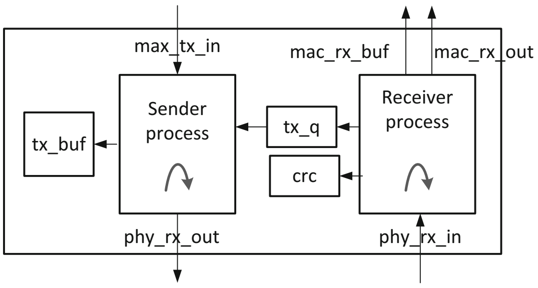


Fig. 7 Example structure of CRC example

Having a sound bottom–up component is very important in the HLS flow for one reason: QOR refinement. Design engineers will often refine and fine-tune the data-path functions to get the best area/timing result possible out of the HLS tools. This is best done when the abstraction is sound and the context in which the function is used is robust to the latency and throughput changes of the data-path function.

3.3 Merging global and local coverage

The unification of the local coverage of the verification of the data-path function along with the coverage from the verification of the control model (simulation or formal) is how the DV team can achieve verification closure.

4 Illustrative example

In this section, we present an illustrative example of a link layer subsystem, interfacing media access control (MAC) and physical (PHY) layers. The subsystem receives messages from the MAC layer and is required to transmit them to the physical layer and vice-versa. In so doing, it uses a cyclic redundancy check (CRC) error detection mechanism that adds the CRC code to messages before sending them to the physical layer. Figure 7 illustrates the structure with the sender and receiver processes and Listing 2 shows the SystemC code for the first model of the receiver process. (While we encapsulate all signal-level communication through the functional `put()`/`get()` interfaces for simplicity, all the underlying signals are visible and used as the variable domain for the property verifications.)

This receiver communicates with the sender through the `tx_q` message buffer. When messages are received and deemed correct, an “ack” message is sent, but only every *N* messages. When the receiver receives the “ack” message, it instructs the sender to clear its retry buffer. The receiver process will request the sender process to send a retry request if a packet has an error. The sender will enter a “go-back-*N*” pattern, starting from a specific message that is in the retry buffer. In the first model, the non-deterministic (or random) signal `nd1` is used to model random packet errors.

4.1 First refinement: properties of global interaction

With the first model, the high level (global) properties to be verified are the required cooperation of both the sender and the receiver processes: this is where the global concurrency is verified. In particular, we check the following properties:

Listing 2 First model of receiver process in link layer

```
void rx_process() {
  reset_interfaces();
  bool drop_until_retry_ack = 0;
  wait();
  while (1) {
    MSG msg = phy_rx_in−>get();
    if (drop_until_retry_ack) {
      if (msg.type==RETRY_ACK) {
        drop_until_retry_ack = 0;
      }
    } else {
      if (msg.type == ACK) {
        tx_q−>put(FLUSH_TXBUF);
      } else if (msg.type==RETRY) {
        tx_q−>put(GO_BACK_N);
      } else {
      bool b = error_check(msg); // semantic stub
        if (b==0) { // message has error
          tx_q−>put(RETRY);
          mac_rx_buf−>clear();
        } else { // message is good
          mac_rx_buf−>put(msg);
          if (msg.last) {
            mac_rx_out−>put(RCV_DONE);
          }
        }
      }
    }
  }
}

void error_check(msg) {
  while (nd_delay.read()) wait(); // random pause−done delay
  return nd_result.read(); // return random good or bad.

}
```

– *Property 1* when a retry-request is received, a retry-ack is sent back, and
– *Property 2* when an ack is received, the sender clears the retry-buffer.

The control-flows of both sender and receiver processes are encoded to cooperate in exercising these scenarios: the process checks if a message passes the check, and if not, it sends a retry request through the sender buffer. Then, it ignores all other incoming messages until it gets an acknowledgement for the retry request.

To test these properties, a representation in the design model of the error check itself and even of the message handling mechanism would be irrelevant, all that is needed is a representation of the possible outcomes: there is some delay and then it is determined that the packet is either *good* or *bad*. The model derived for this property is a semantic "stub" function `error_check()` whose latency (relative to other parallel design actions) is determined by a nondeterministic choice of $\{pause, done\}$ for latency, while the possible error checking function returns of $\{good, bad\}$ likewise are modeled by a nondeterministic choice.

If we want to assure the eventual correctness of these properties, then fairness constraints would be required, specifically that latencies ("pauses") are not infinite. Alternatively, the properties could be recast as safety properties, requiring, for example, that a retry-ack is never sent back unless a retry-request is received, and for the second property, that the sender never clears the retry-buffer unless an ack is received.

4.2 Second refinement: properties in scope of individual processes

The second-level of refinement is to specify the usage of the CRC mechanism for error checking, defining its datatype interface and semantic stubs. A more local design specification property (lower in the design specification hierarchy) may govern local message handling for the CRC. To test this property, one would need a stub for this same function that provided a bit-level representation of the associated data path (a new SystemC process for the computation), but this function model stub may still abstract the actual CRC computation to an empty call with nondeterministic latency and returns, as before.

These changes are listed in Listing 3, where we redefine the body of the `error_check()` function to communicate with the new CRC block that will just return random values with random delay at this level of abstraction. Using this refinement, we can verify the second layer properties:

– *Property 3* when CRC check fails, a retry request is passed to the sender process.

As with Properties 1 and 2, this property would require a fairness constraint on latencies; alternatively, it could be recast as a safety property that requires that a retry request is never passed to the sender unless the CRC check failed.

Listing 3 Refinement for CRC datatypes and stubs

```
void error_check(msg) {
  crc−>put(msg.data, msg.crc);
  return crc−>get(); // get result
}

// Add a new process calling this function:
void compute_crc(DATA data, CRC code) {
  while (nd_crc_delay.read()) wait(); // random delay
  return nd_crc_result.read(); // return random 1 or 0.
}
```

4.3 Third refinement: reducing global to local latency budgets

The third layer of our design specification is about the "latency budget": ensuring that responses are sent within the desired time budget. For this, designers must synthesize the stub models with their specific latencies, and explore which combinations of stub latencies will satisfy the global schedule. These too can be expressed as constraints on model events that impose the required temporal orders. This requires micro-architecture architectural analysis and a strong sense of the realistic implementation budgets for the stubs. Once the temporal constraints are established, it is time for the data-path function verification, as well as verifying the correctness of any fairness constraints that had been imposed on latencies.

4.4 Fourth refinement: data-path function verification

The fourth and lowest-level design specification property in our example stipulates that the CRC implementation conforms to a stated algorithm (correctly divides by the generator), or it may simply state that for a test-bench word generator that generates both correct and corrupted words, the implementation correctly distinguishes between them. For this, the complete SystemC code of the `compute()` function would be used in the verification (no abstraction), so verification of this property would be deferred to the lowest level. An example of the data-path function to be verified is shown in Listing 4.

Listing 4 Leaf refinement for CRC data-path function

```
bool compute_crc (MSG m) {
  sc_bigint<72> datain = m.data;
  sc_uint<8> dataout = 0;
  sc_uint<8> dataout0 = 0;
  for (unsigned int i=0; i<72; i++) {
    dataout0[0]= dataout[7] ^{ } datain[i];
    dataout0[1]= dataout[0];
    dataout0[2]= dataout[1] ^{ } dataout0[0];
    ...
    dataout0[7]= dataout[6] ^{ } dataout0[0];
    dataout= dataout0;
  }
  return (m.crc == dataout); }
```

The designer can then use the microarchitecture command of the HLS tool to unroll the loop, and the HLS tool can optimize the logic to get a parallel computation into efficient RTL and also to meet the latency budgets for implementation.

4.5 Discussion

Although this flow makes it appear as though the designer must encode *n* separate models of the same design for *n* levels of abstraction, it is critical to the usability of the methodology that this is not in fact the case. Indeed, if the methodology entailed writing distinct models for each level of abstraction–in the case of the above example, four separate models for the CRC-calling function, then the design process would quickly become imponderable–it is hard enough to code one design! Coding it several times for the several levels of abstraction would not be considered acceptable.

With the hierarchical design flow advocated here, the design is coded only once. The various levels of abstraction are derived from the *order* of coding. For example, conventionally one would code the CRC-calling function monolithically, integrating its control-flow, message-handling and the CRC computations in a single flat piece of code. In the hierarchical design flow proposed here, one first codes the function's control-flow. This first part of the coding would entail writing mainly the function's semantic stubs with its arguments and returns, and its communication primitives for the numerical computation to be applied to the input word passed by the receiver process. The details of the computation would be deferred until later in the design flow. In order to support verification that includes this high-level representation of the CRC-calling function, its latency and its return would be chosen nondeterministically. In fact, the coding of these choices could be automated, based on the return type of the function and the designation of the code as a "stub". With this stub, properties could be tested that do not depend on the details of how the CRC computations are performed, but only on the return values of the CRC.

Next, the CRC-calling function code would be augmented to add a mechanism for message extraction. It is important that this be an augmentation (or semantic refinement), not a rewriting: new code is added to the original code, defining a code refinement. At this level of abstraction the design could be tested to ensure that the message is properly handled, and this would entail a bit-level representation of the associated data paths. This refinement could be implemented by replacing the higher-level function's abstract data type for *message* by the appropriate word type. Thus, the new model is derived from the previous model by adding code to the previous model.

Eventually, coding is completed by fully expanding data, finally supporting low-level local checks such as the correctness of the CRC. Thus, the CRC-calling function is coded only once, but in a different order than conventionally. Considering a design with many such functions, at first the "high-level" parts of all the functions are coded and tested; next, each function is augmented, writing more of its code, as appropriate for properties to be tested at the next level of abstraction. And so on, until all the coding of all the functions is complete. Each successive model starts with the higher-level control model and expands it, until the coding is complete.

Importantly, this methodology supports allowing the hardware designer to verify the compute functions as soon as their coding is complete, and then progress directly to work on timing/latency goals, in a mix of top–down and bottom–up fashions, where the design architects can verify, analyze and improve the higher-level more global model.

5 Related work

There have been many related proposals over the years exemplified by the methodologies used in Esterel [9,10], StateCharts [11,12] the hierarchical models of Alur [13–15], and the relaxed timing models of Gajski [16–18], where high-level models have been used to guide the verification flow. In all of these cases there is an abstraction hierarchy based upon a separation of a design's control-flow and data-flow.

In his Esterel methodology, Berry has long argued for coding control before data, and parameterizing data paths, but the Esterel methodology does not allow for nondeterminism. Esterel thus supports data type abstraction, but without nondeterminism, the behavioral data path abstraction that we advocate here cannot be supported. Their CRP methodology [19] is a step in this direction, but because of its underlying CSP semantics, it is too restrictive for abstracting data paths in concurrent hardware and getting good QOR in the RTL is a challenge.

StateCharts provides a hierarchical abstraction framework that has been given many different semantics. In principle, it could be given the semantics described here and utilized in the manner that we have proposed, but to our knowledge this has never been done. Alternatively, the Unified Modeling Language (UML) provides multiple graphical notations to specify software systems, and there has been much promising work using the class diagrams and message sequence charts to specify properties and generate stubs models for further software development [20–22] and in the broadening field of model-based design [23].

Alur's hierarchal methodology likewise could support the framework that we propose here, but to our knowledge it never has been applied in this way. Gajski with his team describe new timing abstractions based on the abstractions found in the OSCI SystemC and TLM reference manuals and generally accepted refinement methodologies [24]. The resulting models are described as being loosely-timed, approximately-timed and cycle accurate. While this is related to the idea of the nondeterministic transaction latency, it is unclear if the semantic basis for these abstractions can provide a sound connection to the RTL. The work described in [25–27] uses a high-level model to guide verification, but as it is not formally related to the implementation, there is no way to "take credit" for their high level verification during their RTL verification. Also, cycle-equivalent abstractions do not support the more powerful behavior abstractions and thus verification acceleration that we support through nondeterminism.

There is also promising work on code coverage at the SystemC level [28–31] and techniques to leverage that to improve verification, however, none of these address the abstraction soundness and semantic gap with the implementation models.

To our knowledge, in none of these abstract schemes can high-level verification actually be used to replace RTL verification in a sound manner. The theoretical basis for the methodology presented here was described by the second author in [1]. However, the application of this to a HLM and the presentation of this as a means for "taking credit" for behavior verification during RTL verification is new.

6 Conclusion and future work

We have presented a top–down/bottom–up design methodology that supports verification of system-level properties in an abstract high-level model in a manner that avoids the necessity of reverifying these properties in the RTL. In this way the design development and verification teams can "take credit" for HLM verification in a HLS framework. The top–down methodology is hierarchical and supports acceleration of verification through early debug of controllers before design data-paths are coded and before the design becomes cumbersome to correct or modify, thus retaining all the benefits of utilizing a HLM, including its transparency and relative compactness. The benefit of a sound abstraction is that the properties verified on the abstract model do not have to be re-verified on the refined model (with more details).

The methodology entails a significant change in the conventional flow, requiring the coding of control before data-path. However, this coincides with the architectural design view, and by accommodating it in this manner, aligns the development flow with the design flow.

This is the way design development would naturally evolve, were it not for the technicality that control elements need to refer to data-path elements that thus seemingly inevitably need to be developed first. We have circumvented this technical impediment by replacing the required data-path elements with semantic stubs. In order to be able to utilize these stubs for system-level verification, the stubs are endowed with an interface that semantically abstracts the I/O of the corresponding data-path element.

This methodology supports both formal and simulation-based verification, and any mix of the two. Naturally, formal verification would best be used to verify control properties and system-level global behavior, while simulation and word-level formal verification could be used to verify more local data-path properties.

There are many challenges in fully deploying this methodology. The systematic approach for coverage closure, the ramp-up costs and technology transfer effort may be considerable because it inverts the conventional coding flow that codes data-path before control.

The proposed design/development flow alignment provides an easy entrance for the incorporation of formal verification, and this would provide a very significant enhancement to verification closure.

The methodology described here could be applied to portions of a design as well as to an entire design. The former could limit technology transfer costs, as one could start with a small expert team and then expand the methodology as expertise grows.

The utilization of semantic stubs provides a handy means for a verification group to experiment with the methodology: a little effort yields a little acceleration; a little more yields a little more. Therefore, the ramp-up costs and technology transfer effort can be limited, and once ready with the formal verification expertise to handle more flexible abstractions, the team can move to a full top–down flow. The bottom–up methodology can be gracefully introduced into the design development flow, where the amount of verification acceleration is proportional to the amount of effort applied to writing HLM models, with the resulting benefit that all cover goals met with HLM verification are "virtually" met during RTL verification and thus need not be re-verified in the RTL.

Nonetheless, work remains to establish concrete guidelines for applying this methodology in a standard design flow. Although many have independently applied part or all of this methodology successfully in commercial designs (including, for the second author, one complete design), a few comprehensive case studies would go a long way to illuminate the methodology for routine utilization. One could imagine the development of a "cook book" of supporting strategies. This could be very useful and could pave the way to broader utilization.

Acknowledgements We thank the reviewers for their very helpful suggestions.

References

1. Kurshan RP (1994) Computer-aided verification of coordinating processes. Princeton University Press, Princeton
2. Kurshan RP (2008) Verification technology transfer. In: 25 Years of model checking, LNCS no. 5000, Springer, pp 46–64
3. Bloem R, Jacobs S, Khalimov A, Konnov I, Rubin S, Veith H, Widder J (2016) Decidability of parameterized verification. ACM SIGACT News 47(2):53–64
4. Konnov I, Kotek T, Wang Q, Veith H, Bliudze S, Sifakis J (2016) Parameterized systems in BIP: design and model checking. In: Desharnais J, Jagadeesan R (eds) 27th International conference on concurrency theory (CONCUR 2016), volume 59 of Leibniz international proceedings in informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp 30:1–30:16
5. Bloem R, Jacobs S, Khalimov A, Konnov I, Rubin S, Veith H, Widder J (2015) Decidability of parameterized verification. Synth Lect Distributed Comput Theory 6:1–170
6. Aminof B, Kotek T, Rubin S, Spegni F, Veith H (2014) Parameterized model checking of rendezvous systems. In: Baldan P, Gorla D (eds) CONCUR 2014: concurrency theory, volume 8704 of lecture notes in computer science. Springer, Berlin, Heidelberg, pp 109–124
7. John A, Konnov I, Schmid U, Veith H, Widder J (2013) Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In: FMCAD, pp. 201–209
8. Berry G, Kishinevsky M, Singh S (2003) System-level design and verification using a synchronous language. In: ICCAD '03: proceedings of the 2003 IEEE/ACM international conference on Computer-aided design. IEEE Computer Society, Washington, DC, USA, p 433
9. Berry G, Gonthier G (1992) The Esterel synchronous programming language: design, semantics, implementation. Sci Comput Program 19(2):87–152
10. Berry G (2000) The foundations of Esterel. MIT Press, Cambridge
11. Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8(3):231–274
12. Harel D, Naamad A (1996) The STATEMATE semantics of statecharts. ACM Trans Softw Eng Methodol 5(4):293–333
13. Alur R, Henzinger TA (1999) Reactive modules. Form Methods Syst Des 15(1):7–48
14. Alur R, Grosu R (2000) Modular refinement of hierarchical reactive machines. In: POPL '00: proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM Press, New York, NY, USA, pp 390–402
15. Alur R, Grosu R (2004) Modular refinement of hierarchic reactive machines. ACM Trans Program Lang Syst 26(2):339–369
16. Abdi S, Gajski D (2004) System-level verification with model algebra. CECS, UCI, Technical Report 04-29
17. Abdi S (2005) Functional verification of system-level model refinements. Ph.D. dissertation, University of California, Irvine
18. Abdi S, Gajski D (2006) Verification of system-level model transformations. Int J Parallel Program 34(1):29–59
19. Berry G, Ramesh S, Shyamasundar RK (1993) Communicating Reactive Processes. In: POPL

20. Kollmann R, Gogolla M (2001) Capturing dynamic program behaviour with UML collaboration diagrams. In: Proceedings fifth European conference on software maintenance and reengineering, Lisbon, pp 58–67
21. Cartaxo EG, Neto FGO, Machado PDL (2007) Test case generation by means of UML sequence diagrams and labeled transition systems. In: 2007 in Proceedings of IEEE international conference on systems, man and cybernetics, Montreal, pp 1292–1297
22. Broy M, Krüger I, Meisinger M (2004) Automotive software-connected services in mobile networks: first automotive software workshop, ASWSD 2004, San Diego, CA, USA, January 10–12, 2004, Revised selected papers, Broy M, Krüger I, Meisinger M (eds), Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2006
23. Seiter J, Wille R, Kahne U, Drechsler R (2014) Automatic refinement checking for formal system models. In: Proceedings of the 2014 forum on specification and design languages (FDL), Munich, pp 1–8
24. Groetker T, Liao S, Martin G, Swan S (2002) System design with SystemC. Kluwer Academic Publishers, Dordrecht
25. Flaisher A, Gluska A, Singerman E (2007) Case study: integrating fv and dv in the verification of the intel core 2 duo microprocessor. In: FMCAD ’07: proceedings of the formal methods in computer aided design. IEEE Computer Society, Washington, DC, USA, pp 192–195
26. Beers R (2008) Pre-RTL formal verification: an intel experience. In: DAC ’08: proceedings of the 45th annual design automation conference. ACM, New York, NY, USA, pp 806–811
27. Gluska A, Libis L (2009) Shortening the verification cycle with synthesizable abstract models. In: DAC ’09: proceedings of the 46th annual design automation conference. ACM, New York, NY, USA, pp 454–459
28. Grobe D, Peraza H, Klingauf W, Drechsler R (2008) Measuring the quality of a SystemC Testbench by using code coverage techniques. In: Embedded systems specification and design languages: selected contributions from FDL’07. Springer, Netherlands, pp 73–86
29. Herber P, Glesner S (2013) A HW/SW co-verification framework for SystemC. ACM Trans Embed Comput Syst 12(1s):61
30. Junior AD, da Silva DJC (2007) Code-coverage based test vector generation for SystemC designs. In: IEEE computer society annual symposium on VLSI (ISVLSI ’07), Porto Alegre, pp 198–206
31. Lin B, Yang Z, Cong K, Xie F (2016) Generating high coverage tests for SystemC designs using symbolic execution. In: Design automation conference (ASP-DAC) 2016 21st Asia and South Pacific, pp 166–171