Appears in the Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC-55), Dec 2021

UMOC: Unified Modular Ordering Constraints to Unify Cycle- and Register-Transfer-Level Modeling

Shunning Jiang, Yanghui Ou, Peitian Pan, Christopher Batten
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{sj634, yo96, pp482, cbatten}@cornell.edu

***Abstract*—We propose unified modular ordering constraints (UMOC), a novel approach that seamlessly unifies method-based cycle-level (CL) modeling and signal-based register-transfer-level (RTL) modeling. Motivated by the challenges in state-of-the-art CL modeling methodologies and existing CL/RTL composition attempts, UMOC successfully breaks the trade-off between model fidelity and scheduling modularity for CL modeling and provides seamless composition of CL and RTL models. Instead of requiring the designer to specify the global intra-cycle ordering of hardware processes, UMOC eliminates this burden using implicit local ordering constraints of RTL signals and explicit local ordering constraints of CL methods. We implement and evaluate UMOC in PyMTL3, a state-of-the-art open-source Python-based hardware modeling framework.**

I. Introduction

In response to the growing register-transfer-level (RTL) design effort for modern systems-on-chips (SoC) and the increasing heterogeneity in these SoCs, computer architects have been leveraging domain-specific cycle-level (CL) simulators (CPU [3, 18], memories [16], GPU [2], and on-chip networks [1]), and general-purpose CL modeling frameworks [6, 13, 14], rather than faithfully constructing RTL netlists from the very beginning. Even though CL models include less hardware detail and usually cannot be converted to hardware, the faster simulation speed and easier modification/enhancement is crucial to the early design-space exploration phase. The approximate timing behaviors, combined with analytical area/energy/timing models [11], provide valuable insights to help make first-order design decisions and hence drastically reduce the time spent later in the RTL development phase. After the CL design-space exploration phase, instead of moving directly from a complete CL model to a complete RTL implementation, the ability to seamlessly mix and match RTL models with CL models brings significant productivity benefits. Gradually swapping CL blocks for freshly developed RTL blocks makes it easier to: (1) maintain the integration tests, end-to-end tests, and performance regressions, and (2) steadily improve the model fidelity of the whole design. Prior research attempts to unify the cycle-level descriptions and RTL generation for specific hardware domains (e.g., architectural description languages for processors [4, 7]). *This paper focuses on general-purpose CL/RTL modeling and composition mechanisms.*

Unlike RTL modeling's well-established discrete-event simulation semantics, the inter-cycle and intra-cycle semantics are different across different CL simulators. Commonly used CL inter-cycle mechanisms include: (1) discrete-event simulation that maintains an event queue to automatically advance the timestamp and trigger designer-scheduled events of hardware processes [1, 3, 14, 18], and (2) cycle-by-cycle simulation which essentially assumes all hardware processes are recurringly triggered at every rising clock edge [2, 6, 8, 13, 16]. When several hardware processes are triggered at the same timestamp in both cases, the intra-cycle mechanism has to decide the order of execution. *This paper focuses on intra-cycle mechanisms.* The most commonly used CL intra-cycle mechanism is designer-specified global ordering of hardware process invocations for modeling combinational/sequential behaviors. However, global intra-cycle ordering makes it challenging to achieve model fidelity *and* scheduling modularity at the same time. State-of-the-art mechanisms for composing CL and RTL models are ad-hoc and only enable heterogenous compositions across different models of computation, due to the intra-cycle semantic gap between CL and RTL modeling. As elaborated in Section II, we identify two major challenges in state-of-the-art CL simulators/frameworks and attempts to compose CL and RTL models: (1) the trade-off between model fidelity and scheduling modularity in CL modeling; (2) seamless composition of CL and RTL models.

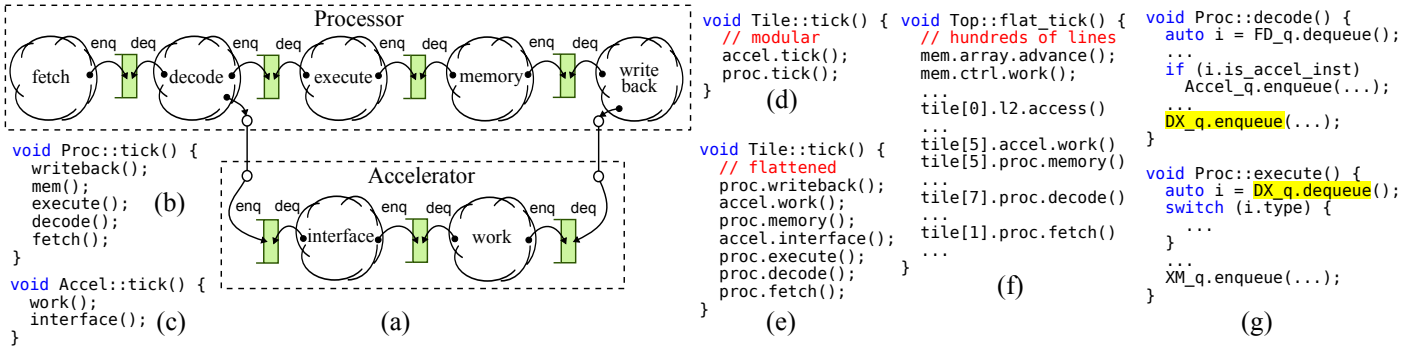
In this paper, we introduce a novel intra-cycle modeling mechanism that unifies method-based CL modeling and signal-based RTL modeling to solve these challenges. Unified modular ordering constraints (UMOC) provide a unified view for general-purpose CL and RTL modeling and enable automatically scheduling all the CL/RTL processes with designer-specified (CL) or inferred (RTL) *local* constraints without manually specified global intra-cycle ordering of hardware processes. Section III discusses the key idea and foundation of UMOC. UMOC can be implemented in any unified CL/RTL modeling framework (e.g., SystemC [14]). We implement UMOC in PyMTL3 [10], a recently developed open-source Python-based hardware modeling framework. We explain the implementation in Section IV. Section V includes two case studies on how UMOC with PyMTL3 enables accurately composing CL/RTL processors and CL/RTL checksum accelerators, and a bigger CL/RTL manycore system.

This paper makes the following contributions: (1) we identify two key challenges to CL modeling and CL/RTL composition; (2) we propose unified modular ordering constraints (UMOC) to address these challenges; and (3) we explain our UMOC implementation in PyMTL3, a state-of-the-art hardware modeling framework.

II. Related Work and Motivation

In this section, we identify two key challenges to CL modeling and CL/RTL composition, along with the corresponding related work.

Challenge #1: Trade-off between model fidelity and scheduling modularity in cycle-level modeling – Cycle-level simulators [1–3, 16, 18] usually improve the model fidelity against the target architecture by specifying the intra-cycle total ordering of calling hardware processes to model the desired pipeline/combinational behavior. Figure 1(b–c) shows an example of a C++ simulator modeling the composition of one processor and one tightly coupled hardware accelerator in Figure 1(a) using reversed invocation order for pipeline behavior. Note that invoking processor and accelerator schedules as blackboxes at the top level as shown in Figure 1(d) harms the model fidelity regardless of the invocation order of `proc.tick()` and `accel.tick()`. Essentially, simply composing two modular "pipelines" and concatenating their execution schedule gives up the possibility to interleave hardware processes in these pipelines and can create a behavior mismatch against




```
void Proc::tick() {
  writeback();
  mem();
  execute();
  decode();
  fetch();
}
```

```
void Accel::tick() {
  work();
  interface();
}
```

```
void Tile::tick() {
  // modular
  accel.tick();
  proc.tick();
}
```

```
void Tile::tick() {
  // flattened
  proc.writeback();
  accel.work();
  proc.memory();
  accel.interface();
  proc.execute();
  proc.decode();
  proc.fetch();
}
```

```
void Top::flat_tick() {
  // hundreds of lines
  mem.array.advance();
  mem.ctrl.work();
  ...
  tile[0].l2.access()
  ...
  tile[5].accel.work()
  tile[5].proc.memory()
  ...
  tile[7].proc.decode()
  ...
  tile[1].proc.fetch()
  ...
}
```

```
void Proc::decode() {
  auto i = FD_q.dequeue();
  ...
  if (i.is_accel_inst)
    Accel_q.enqueue(...);
  ...
  DX_q.enqueue(...);
}
```

```
void Proc::execute() {
  auto i = DX_q.dequeue();
  switch (i.type) {
    ...
  }
  ...
  XM_q.enqueue(...);
}
```

Figure 1. **Modeling a Cycle-Level Processor/Accelerator Tile –** An example abstracted from real-world simulator code: (a) the pipeline structure and composition of a five-stage processor and a two-stage tightly coupled accelerator where the accelerator request is sent out at `decode` and the response is accepted at `writeback`; (b–c) the tick methods of `Proc` class and `Accel` class, both of which model pipeline behavior; (d) the modular tick method of `Tile` that calls the tick of `Proc` and `Accel`; (e) the flat tick method that directly calls the hardware logic inside `Proc` and `Accel` for more accurate performance modeling; (f) the hypothetical flat tick function of a complex design that models the performance accurately; (g) `Proc::decode` and `Proc::execute` communicate through buffer `DX_q`.

the target architecture. This is a module-level cyclic inter-dependency that the modular tick approach cannot break. Admittedly, the designer should be able to break the modularity to improve performance fidelity as illustrated in Figure 1(e) to resolve the module-level dependency. However, to the best of our knowledge, we have rarely seen any simulator that abandons scheduling modularity, simply because it is hard to maintain a flattened top-level schedule of a complex hardware block (see Figure 1(f)), especially during incremental development. gem5 [3] relies on designer-marked single-integer priority on each hardware process and decides the global intra-cycle ordering by sorting the events based on priority. Specifying incorrect priority will lead to unexpected and profound performance bugs such as erroneous combinational behavior between two decoupled modules, and it is impossible to report any mistake during scheduling under this scheme.

We conclude that the state-of-the-art CL modeling approaches rely on designer-specified *global* intra-cycle ordering of hardware processes, which makes it challenging to attain scheduling modularity *and* performance fidelity at the same time.

Challenge #2: Seamless composition of CL and RTL models – Several general-purpose modeling frameworks have provided first-class support for composing cycle-level models and RTL models. Cascade [6] is a CL modeling framework which provides RTL-like register elements and combinational updates as modeling primitives. Cascade supports composing cycle-level models written in C++ with Verilog by exporting the CL model as a standalone C module and importing it inside a Verilog module using Verilog Procedural Interface (VPI). However, the top-level simulation driver is the Verilog simulator. SystemC [14] provides a unified environment in C++ for CL and RTL modeling. However, SystemC primitives for transaction-level modeling (TLM) are often used for functional verification rather than accurate performance modeling. For example, it is impossible to model intra-cycle behavior going through RTL–CL–RTL if TLM channels are used as interfaces, which makes it hard to do fine-grained intra-cycle CL/RTL composition. PyMTL [13] also unifies CL/RTL modeling in Python by instantiating port-based RTL interfaces inside CL models and wrapping RTL interfaces with CL buffers with enqueue/dequeue methods for CL processes to call. PyMTL supports event-driven semantics for RTL models, but the designer has to manually call the CL processes in a total order like Figure 1(b-f). Hence, PyMTL fails to close the CL/RTL semantic gap.

There are also ad-hoc attempts [5, 12] to compose established CL and RTL simulators. PAAS [12] supports coarse-grained composition of Verilog RTL accelerators with gem5 CPU and memory models by using linux `/dev/shm` shared memory to exchange data between gem5 and a Verilator-compiled [17] C++ simulator.

We conclude that previous attempts to compose CL and RTL models are ad-hoc and design-specific at a coarse granularity. As far as we are aware, no prior work has provided a seamless composition of CL and RTL models using a unified model of computation.

III. Unified Modular Ordering Constraints

In this section, we describe unified modular ordering constraints (UMOC), a novel *intra-cycle scheduling mechanism* to unify CL/RTL modeling which tackles the two challenges in Section II.

A. RTL Scheduling with Implicit Constraints

If behavioral RTL process A writes signal x and B reads x, traditional HDL simulators will infer this sensitivity and dynamically schedule B to execute whenever A modifies x. Inspired by previous work on statically scheduling RTL processes [6, 9, 15], we propose to use the notion of ordering constraints to implicitly deduce the relationship between block A and B as follows.

$$\left.\begin{array}{c} x \text{ is a combinational wire} \\ A \text{ writes signal } x,\ B \text{ reads signal } x \end{array}\right\} \Longrightarrow \begin{array}{c} A \text{ precedes } B \\ (A < B) \end{array}$$

The key observation here is that even though x is merely a *local* variable w.r.t. A and B, the ordering between A and B is later used by the scheduler *globally* to determine the final execution order of all RTL processes in the design. This is because in a hierarchical RTL model, an RTL module exposes ports to the parent module which are connected to signals in other modules. All the connected signals are essentially the same signal, and hence the preceding relationship of any two faraway combinational RTL processes can be established without exposing any details inside the module, which preserves the *modularity*.

B. CL Scheduling with Explicit Constraints

For CL modeling, we also want to reduce the burden on designers by propagating local ordering constraints. However, there is no signal in

CL models, as CL models deal with high-level data structures. We observe that CL processes still need to communicate via buffers that expose *methods* for CL processes to call (similar to SystemC `sc_fifo`). For example, Figure 1(g) shows that `decode` enqueues a message to `DX_q` and `execute` dequeues the message (using a queue handles the back pressure from a later pipeline stage). The reversed order in Figure 1(b) guarantees that `execute` is called before `decode` in every clock cycle, which means ***`dequeue`*** *of the buffer is always called before* ***`enqueue`***. Thus, whatever `decode` enqueues to the buffer will only be dequeued by `execute` in the next cycle to model pipeline behavior. Conversely, calling `decode` before `execute` results in combinational bypass behavior.

From the above observation, we further discover that specifying the global ordering (Figure 1(b)) essentially controls the order of calling `enqueue` and `dequeue` of the buffers in a cycle. **Can we specify the ordering inside the buffer directly so that the order between the functions that call `enqueue` and `dequeue` can then be inferred globally?** The answer is positive, and the deductive process with explictly specified local constraints between `enqueue` and `dequeue` methods is shown below. Simply flipping the local ordering constraints allows the designer to model combinational behavior *with the same set of methods* without changing anything else.

$$\left.\begin{array}{c}\texttt{q.dequeue}\ \textbf{precedes}\ \texttt{q.enqueue}\\ A\ \text{calls}\ \texttt{q.dequeue},\ B\ \text{calls}\ \texttt{q.enqueue}\end{array}\right\} \Longrightarrow \begin{array}{c}A\ \text{precedes}\ B\\ (A<B)\end{array}$$

C. Achieving Fidelity and Modularity At Once

We use the processor/accelerator example in Figure 1 to explain how Challenge #1 in Section II can be fully addressed by explicit ordering constraints. We first create a pipeline queue which specifies { `dequeue < enqueue` }. Then we instantiate it between the stages in `Proc` and `Accel`. The global scheduler can automatically deduce the reversed invocation order of Figure 1(b–c) without the designer-written tick methods. To accurately model the communication between the processor and the accelerator in Figure 1(a), we also need to put two queues inside `Accel` as the communicating buffer for `Accel::work` and `Proc::writeback`, and for `Proc::decode` and `Accel::interface`. For the former pair, since `Accel::work` and `Proc::writeback` are not in the same module, we need to expose the "pointer" of the `dequeue` method from `Accel` to the parent module `Tile` (similar to SystemC `sc_export`) and pass it into `Proc` so that `Proc::writeback` actually calls the `dequeue` method of the queue in `Accel`. The latter pair can be handled similarly by exposing the `enqueue` method from `Accel`.

The global scheduler then automatically deduces { `Proc::writeback < Accel::work`, `Accel::interface < Proc::decode` }. The designer does not need to write `Tile::tick` and `Top::tick` like Figure 1(d–f) at all. A feasible global schedule is able to achieve the same *model fidelity* as flattened tick functions like Figure 1(e–f). Moreover, the *modularity* is preserved at the same time. `Accel` module now exposes a `dequeue` method and an `enqueue` method to the outside world, which means we can use the accelerator *as a standalone module* to build other systems without knowing any detail inside `Accel`. Any CL process P that calls the exposed `dequeue` automatically results in an ordering constraint $\{P < \texttt{Accel::work}\}$.

D. Unified Directed Graph (UDG)

Creating the Unified Directed Graph – The key to solve Challenge #2 in Section II is to create a unified directed graph (UDG) $G = (V, E)$

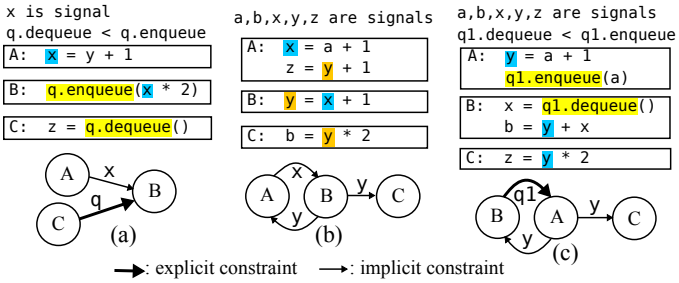


Figure 2. **UMOC Examples** – Code of CL/RTL processes and corresponding unified directed graphs: (a) CL/RTL constraints can co-exist; (b) cycle of RTL processes; (c) cycle of CL processes.

where V includes all the hardware processes and E includes all the implicit/explicit ordering constraints between them. For any mixed CL/RTL design, applying the deductive process in Section III-A and III-B can set up the preceding relationships not only between all pairs of RTL processes and all pairs of CL processes, but also CL and RTL processes. We can deduce two ordering constraints in Figure 2(a): $\{A < B\}$ from signal x and $\{C < B\}$ from { `q.enqueue < q.dequeue` }. Here, B serves as the "glue" between CL and RTL portions of the design by accessing signals and calling methods at the same time. Note that G may contain cycles. UMOC allows the UDG to have cycles among *only combinational RTL processes* and defers the combinational loop detection to the real simulation if the signal values fail to stabilize. However, UMOC does not allow cycles that include any CL process, because CL processes are usually modeled to execute once per clock cycle due to the side effects on high-level data structures. For example, executing process A of Figure 2(c) multiple times may unexpectedly enqueue many elements into $q1$.

Scheduling the Unified Directed Graph for Simulation – We cannot directly reuse canonical event-driven RTL scheduling algorithms for unified CL/RTL scheduling. This is again because CL processes use high-level data structures instead of signal/ports, and CL processes are usually modeled to execute exactly once per cycle (see Figure 1(g)). Essentially, a correct execution of G must guarantee that before executing any CL process, *all* preceding processes should have been executed, and the cycles of preceding RTL processes have stabilized.

If G contains no cycle, i.e., G is a directed acyclic graph (DAG), a topological sort on G will yield a valid serial schedule. In each clock cycle, we can simply enumerate the serial schedule to execute each hardware process exactly once, satisfying the guarantee for CL processes. If G contains cycles, according to classic graph theory, a "cycle" in a directed graph is defined as a strongly connected component (SCC) in which every vertex is reachable from every other vertex. The scheduler can apply classic SCC algorithms to transform G into a DAG G' of SCCs. Each SCC represents a single vertex in G' or a "cycle" in G. Applying a topological sort on G' yields a serial schedule of all the SCCs. During simulation, we execute all the SCCs in the schedule in each clock cycle. For single-node SCCs, we execute the only hardware process. For multi-node SCCs, we need to iteratively execute all the RTL processes until the signals stabilize and report a combinational loop when it fails to converge.

IV. UMOC Implementation in PyMTL3

In this section, we present the UMOC implementation in PyMTL3 [10], an open-source Python-based hardware modeling framework. Note that UMOC can be implemented in other frame-

```python
class RegIncrRTL( Component ):
  def construct( s ):
    s.in_ = InPort (32)
    s.out = OutPort(32)

    s.reg = Wire(32)
    @update_ff
    def seq_reg():
      s.reg <<= s.in_

    @update
    def comb_out():
      s.out @= s.reg + 1
```

(a) RTL RegIncr Unit

```python
class WireIncrRTL( Component ):
  def construct( s ):
    s.in_ = InPort (32)
    s.out = OutPort(32)

    s.wire = Wire(32)
    @update
    def comb_wire():
      s.wire @= s.in_

    @update
    def comb_out():
      s.out @= s.wire + 1
```

(b) RTL WireIncr Unit

```python
class RegIncrCL( Component ):
  def construct( s ):
    s.add_constraints(
      M(s.read) < M(s.write),
    ) # Sequential behavior!

  @method_port
  def read( s ):
    return s.v + 1

  @method_port
  def write( s, v ):
    s.v = v
```

(c) CL RegIncr Unit

```python
class WireIncrCL( Component ):
  def construct( s ):
    s.add_constraints(
      M(s.write) < M(s.read),
    ) # Combinational behavior!

  @method_port
  def read( s ):
    return s.v + 1

  @method_port
  def write( s, v ):
    s.v = v
```

(d) CL WireIncr Unit

```python
class RegIncrCLRTL( Component ):
  def construct( s ):
    s.write = CalleePort()
    s.out   = OutPort(32)
    s.r1 = RegIncrCL()
    s.r2 = RegIncrRTL()
    connect(s.write, s.r1.write)
    connect(s.out,   s.r2.out  )

    @update_once
    def send_to_r2():
      s.r2.in_ @= s.r1.read()
```

(e) CL+RTL Two-Stage RegIncr

```python
class RegIncrRTLCL( Component ):
  def construct( s ):
    s.in_  = InPort(32)
    s.read = CalleePort()
    s.r1 = RegIncrRTL()
    s.r2 = RegIncrCL()
    connect(s.in_,  s.r1.in_ )
    connect(s.read, s.r2.read)

    @update_once
    def send_to_r2():
      s.r2.write( s.r1.out )
```

(f) RTL+CL Two-Stage RegIncr

Figure 3. **PyMTL3 Buffered Incrementer Units** – (a–d) shows the RTL/CL implementations of a registered incrementer and a wire incrementer; (e–f) shows the two possible RTL/CL compositions.

works as well. We implement cycle-by-cycle simulation as the inter-cycle mechanism and UMOC as the intra-cycle mechanism. Leveraging Python's productive features, we implement a set of modeling primitives for the designer to construct CL and RTL models. Then we implement PyMTL3 passes to build and schedule the unified directed graph for simulation. Figure 3 shows six PyMTL3 code examples.

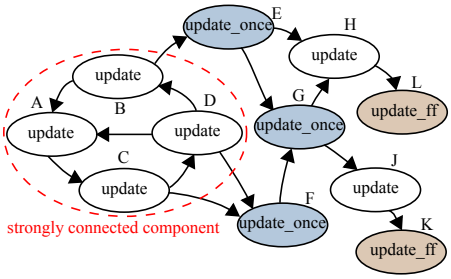
A. Modeling Primitives

Here we explain a minimum set of necessary primitives to simplify the context. Syntactic sugar can be created on top of them to further improve the productivity of hardware designers.

Components – A component is a hardware module that includes RTL and/or CL processes, and child components.

Signals and Value Ports – Implicit ordering constraints are inferred from accesses to signals and value (input/output) ports. Value ports are exposed to the parent component. Normal signals are internal. Connecting signals and value ports associates all connected signals/ports with the same value and hence propagates the implicit constraint outside the component, which is the key to modularity.

Methods and Method Ports – Methods are member functions of a component. Method (caller/callee) ports are exposed to the parent component. The designer explicitly specifies the ordering constraints that involves methods, which will be collected by PyMTL3 during elaboration. Connecting methods and method ports essentially propagates the specified constraints outside the component.



(a) A unified directed graph example

```
flip_registers()
while not stable:
  A()
  C()
  D()
  B()
  check_threshold()
E() # After SCC
F() # After SCC
G() # After E,F
H() # After E,G
J() # After G
L() # After H
K() # After J
```

(b) 1-cycle execution

```
1:  procedure TICK ( top )
2:      flip_registers( top )
3:      for each SCC c in top.schedule do
4:          if size(c) == 1 then
5:              Execute the only block b in c
6:          else
7:              count = 0
8:              while outputs from c does not stabilize do
9:                  for each block b in c do
10:                     Execute b
11:                 count = count + 1
12:                 if count > threshold then
13:                     error("Found combinational loop!")
```

(c) Generated *tick* function

Figure 4. **Scheduling and Simulating a Design –** (a) the corresponding graph of a design with 11 update blocks, four of which form a strongly connected component; (b) one-cycle execution trace of the *tick* function; (c) the generated *tick* function.

Update Blocks – We model hardware processes in PyMTL3 using three types of blocks: `update` for combinational RTL logic, `update_ff` for sequential RTL logic, and `update_once` for CL modeling. All blocks can read/write signals and ports. Any signal/port written by a non-blocking assignment in an `update_ff` block is inferred as a sequential element and not counted in ordering constraint deduction. Hence `update_ff` blocks will not precede any other block. `update_once` blocks can also *call methods and method ports*, and hence are restricted to be executed exactly once in each cycle.

Setting Ordering Constraints – We add an API to PyMTL3 for the designer to specify two types of explicit ordering constraints between (1) methods and (2) methods and update blocks. Figure 3(c–d) shows the constraints set between two methods: `read` < `write` for sequential behavior, and `write` < `read` for combinational behavior.

B. Building the Unified Directed Graph

We implement a PyMTL3 UDG generation pass that generate the UDG $G = (V, E)$ of an elaborated PyMTL3 model. V includes all three types of update blocks, and E includes all the implicit and explicit ordering constraints between those blocks. Figure 4(a) shows an 11-node UDG example.

Implicit Ordering Constraints – We implement a two-step algorithm. First, we leverage Python's introspection features to obtain the abstract syntax tree of each update block, look for read/write variables, and turn each variable name into an actual object using Python's reflection features. If an object is of signal/port type, we associate the object with the update block. The second step enumerates all the signals collected throughout the hierarchy to perform the deductive process in

Section III-A. For each signal x, we add a unidirectional edge $A \rightarrow B$ to the edge set E if block A writes x and block B reads x and A is not an `update_ff` block.

Explicit Ordering Constraints – As Python methods are objects, we apply the same AST-based approach to obtain what methods each `update_once` block invokes. Then, we assemble the invocations with the explicit ordering constraints specified by the designer and perform the deductive process in Section III-B. Specifically, if block A calls method P and block B calls method Q, and the explicit method/method constraint $P < Q$ exists, we add a unidirectional edge $A \rightarrow B$ to the edge set. Likewise, if block A calls method P and there is an explicit method/update constraint $P < B$ between method P and block B, we add $A \rightarrow B$ to the edge set.

C. Scheduling the UDG for Simulation

We implement the scheduling algorithm in Section III-D as a PyMTL3 scheduling pass to condense G into a DAG G' of SCCs (e.g., the "cycle" in Figure 4(a) will become a single vertex in G'), followed by a topological sort on G' to produce a linear schedule. The pass also checks that any non-trivial SCC doesn't contain `update_once` blocks. Otherwise, the designer must remove the interdependencies.

Then, the tick generation pass takes the schedule and creates a *tick* function that simulates for one clock cycle as shown in Figure 4(c). The pass creates a function *flip_registers* for *tick* to call at the rising clock edge to double-buffer all sequential elements that appear in the non-blocking assignments of `update_ff` blocks. All the SCCs in the schedule are then executed. The execution of each SCC is either executing one block or repeatedly executing the `update` blocks until the signals stabilize. If the execution does not converge until it reaches the threshold, a combinational loop is detected. Figure 4(b) shows *tick*'s execution for one clock cycle.

V. Case Studies

We present two realistic case studies to showcase the effectiveness of UMOC. The designs used are all implemented in PyMTL3. The first case study includes a processor/accelerator composition similar to the motivating example in Figure 1, which demonstrates that UMOC can solve the two challenges in Section II. The second case study includes a larger many-core design as evidence for UMOC's ability to handle larger designs with fine-grained CL/RTL compositions for fast design-space exploration during the iterative development process.

A. Processor/Accelerator Composition

We implement a classic 5-stage pipelined RTL RISC-V processor, and a 3-stage pipelined cycle-level RISC-V processor which contains only three `update_once` blocks to approximately model the RTL processor (`fetch`, `decode+execute+memory`, and `writeback`). We expect a little timing difference across CL and RTL processors, as different number of stages lead to different stalling behaviors of read-after-write (RAW) hazards. We also implement RTL and CL Fletcher's algorithm checksum accelerators in PyMTL3. The CL accelerator contains two `update_once` blocks to model the request handling and the actual computation using normal Python functions, where the RTL accelerator implements a fairly complex hierarchical design with eight `StepUnit` instances and a finite state machine. For pure-CL composition, we instantiate cycle-level pipeline queues which already include explicit ordering constraints for the `update_once` blocks in the CL processor and CL accelerator to communicate. Thus we do not need to set *any* constraints in the processor and the accelerator. We are also able to expose and connect the queue methods at the top-level.

Mechanism	Composition	#Cycles	Deviation	Remarks
Event-driven	RTL Proc + RTL Accel	565	-	baseline
UMOC	RTL Proc + RTL Accel	565	0%	same as baseline
UMOC	CL Proc + CL Accel	541	4%	due to 3-stage
Manual P<A	CL Proc + CL Accel	416	26%	modular sub-tick
Manual A<P	CL Proc + CL Accel	416	26%	modular sub-tick
UMOC	CL Proc + RTL Accel	541	4%	same as CL+CL
UMOC	RTL Proc + CL Accel	565	0%	same as RTL+RTL

TABLE I. Results for CL/RTL Proc/Accel Case Study

Table I shows the simulated cycle count of various compositions running the same microbenchmark. The rolling checksum microbenchmark contains a 25-iteration loop, with each iteration sending 3 loads to memory and 6 requests to the accelerator, resulting in a total of 314 dynamic instructions. For the pure RTL composition, event-driven simulation finishes in 565 cycles, and UMOC has exactly the same simulated cycle count. For the pure CL composition, the global schedule automatically generated by UMOC is able to achieve 4% cycle count difference, which is expected as mentioned previously due to the simplified 3-stage processor. To model the "manual modular sub-tick" in Figure 1(b–d), we manually create two tick functions for CL processes inside processor and accelerator. For `P<A`, we invoke processor's tick before accelerators's tick, and `A<P` does the opposite. We verify that the tracing output shows unexpected combinational behavior in both cases in contrast to UMOC. As a result, the simulated cycle count has 26% deviation from the pure RTL composition, much worse than UMOC's 4%.

For mixed CL/RTL cases, we basically insert adapters of "glue" blocks at the CL/RTL boundary. PyMTL3 allows us to create adapters for automatically connecting CL/RTL interfaces, which makes the CL/RTL integration effortless. Simulation results show that the CL processor with RTL accelerator has the same cycle count as CL processor with CL accelerator. Also, RTL processor with CL accelerator has the same cycle count as the pure RTL composition. This confirms that UMOC can provide seamless CL/RTL composition under the same abstraction without losing any model fidelity.

B. Many-Core/Cache/Network Composition

We implement a many-core system that consists of a parametrizable amount of tiles. Each tile contains a parametrizable amount of RV32IMAF cores and data caches, sharing one instruction cache, one integer multiply/divide unit (MDU), and one floating point unit (FPU) via on-chip interconnect networks. Throughout the development process, we extensively use fine-grained CL/RTL mixed compositions enabled by UMOC to facilitate design-space exploration, performance evaluation, and the decision on RTL implementation. The CL models are able to capture the desired cycle-level behavior using UMOC explicit constraints and the scheduling pass. UMOC also enables us to seamlessly integrate existing RTL IP blocks that have been fully tested and prototyped in the past, instead of developing additional CL models. Figure 5 shows the many-core system with a CL main memory. Each block is annotated with the availability of CL, RTL, or both CL and RTL models.

The purpose of implementing the CL multiplier/divider is for quickly studying the performance to decide the type of RTL unit (pipelined or iterative) and the latency/throughput (number of pipeline stages or processed bits per cycle) needed, when shared by multiple processors. After simulating multiple workloads, we decided to implement the iterative divider in RTL because the ratio of div/mod instructions is low. However, we decided to implement radix-four iterative divider so that each div/mod operation takes 16 instead of 32 cycles,

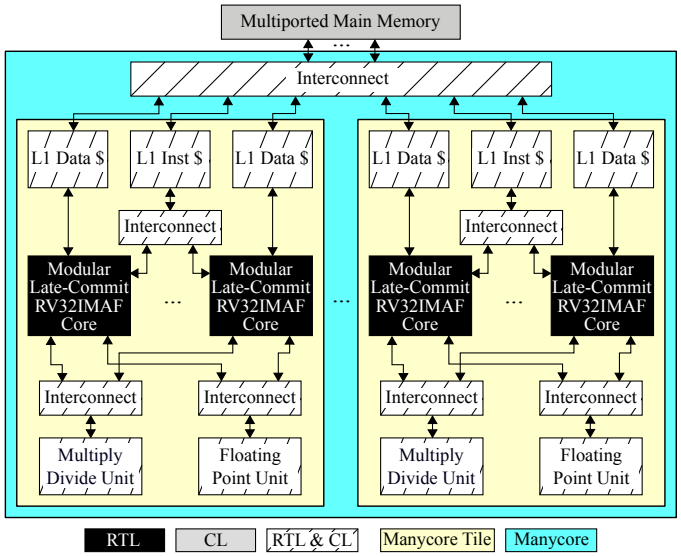


Figure 5. **Tiled many-core with mixed CL/RTL components –** Different colors/patterns show the CL/RTL component availability during the development process. We directly reused the RTL processor, because it was already available prior to the many-core project. We only developed CL model for the main memory, because the main memory is only for testing and verification.

since most division operations are found to stall many subsequent instructions. For the multiplier, we decided to implement a four-stage pipelined multiplier for higher throughput, as some benchmarks contain streams of multiply instructions. The CL models only contain one `update_once` block which processes the request, does the computation, and sends the response to delayed buffers. The user does not need to set any explicit ordering constraints in the multiplier/divider, as appropriate explicit ordering constraints are automatically set when the delay buffers are instantiated with different delays.

As we already developed the RTL processor, developing a CL cache enables quickly exploring the system-level impact of a one-cycle vs. two-cycle hit-latency under different cache sizes and associativities. This influences the parameter selection of different data structures inside the processor. The CL cache model only contains several `update_once` blocks which are responsible for composing requests and responses, which is much simplier than the final RTL cache that consists of tens of different components. As Figure 5 shows, we have a few different on-chip interconnects in this many-core composition. We are able to develop a single CL network with less than two hundred lines of code to guide the decision of each RTL network implementation. The CL model is essentially a crossbar network, but provides the ability to configure the latency between each pair of input/output terminals, and the size of each terminal buffers, which allows CL model to capture the behavior of any complex network topology.

VI. Conclusions

In this paper, we propose a novel approach, unified modular ordering constraints (UMOC), to unify cycle-level and register-transfer-level modeling. UMOC addresses the challenges in the state-of-the-art CL modeling approaches and CL/RTL composition approaches. We demonstrate the feasibility of our approach by implementating UMOC in PyMTL3, an open-source Python-based hardware modeling framework.

Acknowledgments

This work was supported in part by NSF CRI Award #1512937, DARPA POSH Award #FA8650-18-2-7852, a research gift from Xilinx, Inc., and the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, as well as equipment, tool, and/or physical IP donations from Intel, Xilinx, Synopsys, Cadence, and ARM. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

References

[1] N. Agarwal et al. GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2009.

[2] A. Bakhoda et al. Analyzing CUDA Workloads Using a Detailed GPU Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2009.

[3] N. Binkert et al. The gem5 Simulator. *SIGARCH Computer Architecture News (CAN)*, 39(2):1–7, Aug 2011.

[4] A. Chattopadhyay et al. LISA: A uniform ADL for embedded processor modeling, implementation, and software toolsuite generation. In *Processor description languages*, pages 95–132. Elsevier, 2008.

[5] N. Ganjehloo et al. Integrating Cycle Accurate Chisel Models with gem5's System Simulation, 2018.

[6] J. P. Grossman et al. The Role of Cascade, a Cycle-Based Simulation Infrastructure, in Designing the Anton Special-Purpose Supercomputers. *Design Automation Conf. (DAC)*, Jun 2013.

[7] A. Halambi et al. EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. *Design, Automation, and Test in Europe (DATE)*, Mar 1999.

[8] N. Jiang et al. A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2013.

[9] S. Jiang et al. Mamba: Closing the Performance Gap in Productive Hardware Development Frameworks. *Design Automation Conf. (DAC)*, Jun 2018.

[10] S. Jiang et al. PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification. *IEEE Micro*, 40(4):58–66, Jul/Aug 2020.

[11] S. Li et al. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2009.

[12] T. Liang et al. Paas: A system level simulator for heterogeneous computing architectures. *Int'l Conf. on Field Programmable Logic (FPL)*, Sep 2017.

[13] D. Lockhart et al. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture (MICRO)*, Dec 2014.

[14] P. R. Panda. SystemC: A Modeling Platform Supporting Multiple Design Abstractions. *Int'l Symp. on Systems Synthesis (ISSS)*, Oct 2001.

[15] D. G. Pérez et al. A New Optimized Implemention of the SystemC Engine Using Acyclic Scheduling. *Design, Automation, and Test in Europe (DATE)*, Feb 2004.

[16] P. Rosenfeld et al. DRAMSim2: A cycle accurate memory system simulator. *Computer Architecture Letters (CAL)*, 10(1):16–19, 2011.

[17] Verilator. Online Webpage, 2019 (accessed Nov 15, 2019).

[18] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr 2007.