DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Simulation Phases

What are the phases of simulation, and should they be dynamic?

Mark Burton
Qualcomm France SARL
mburton@qti.qualcomm.com

Mark Glasser
glassermark@gmail.com

Karsten Einwich
Coseda Technologies
karsten.einwich@coseda-tech.com

Ramzi Karoui
ZettaScale
ramzi.karoui@zettascale.tech

Abstract — **There are decades of research and development into simulation, yet, while the importance of (increasingly) complex simulation including diverse system components from fluid to electronics is well understood, there seems to be a simple question for which there is no clear answer:**

> ***"What are the phases of simulation?"***

Well, as Benjamin Franklin is often miss-quoted as saying 'Nothing is certain except [birth] death and taxes'. Maybe there-in lies the answer to the question. After all, all we can really be certain of is that simulators will start, run and stop.

In asking this question, and looking into the various methods used by different simulation environments, what has become rather apparent is that, in general, all of them have grown in some way organically, independently, and - probably - with no specific underlying theoretical basis. Our aim here is at least to provide a framework from which future developments can be viewed.

This paper wont give answers, nor code, but draws the conclusion that in the same way that simulation events passed from a static ordering to use a dynamic mechanism, dynamic phasing should be considered for simulators and federations of simulators.

I. Phases

Simulation phasing breaks into three groups of phases: preparation, execution, and termination. Further subdivisions of those groups is somewhat arbitrary, and is based on dependencies.

What are phases, and why have them?

> **The role of the phases is to enforce ordering of operations and to ensure that dependencies are satisfied.**

Before defining terms, lets be clear - everybody disagrees on simulator terminology. Your humble authors were unable to come up with better terms. Defining simulation phases is similarly difficult. We might wish to think about simulations containing multiple (heterogeneous) components that are in different "Phases". Lets lean on our friends in the physics department, a simulation - like a material - may contain many different components, but together lets say they are all at one "State"; though, individually they may be in different "Phases". In short, lets use "Phases" when referring to the state of a single homogeneous simulation environment (or component of a wider "federated" simulation), and "State" when referring to a heterogeneous overall (federated) simulation. Please note, this terminology is arbitrary.

Different simulation environments use different terminology here, so whatever naming convention we choose may not be universal. Simulation environments with solvers are often said to have state (in the solver) for instance.

Another common distinction is to consider the simulator itself to have phases and individual models inside the simulator to have state. The distinction between model and simulator is one of the the focuses of this paper. However for this paper we will use model and simulation state to distinguish between them.

More formally, others have talked about “tags” [1] (stepping away from controversy perhaps). Tags relate to modelled time points, rather than simulation phases. If the reader is so inclined, an abbreviation of this paper could be to simply point out that there are tags which span multiple (or indeed all) models, and those tags are often associated with the simulator phases. Different simulators then treat those tags effectively as a model of computation in their own right. This paper makes the following claims:

1) Simulation phases can be treated as tags in that they adhere to a model of computation in the same way.
2) In general, a tag system defines a dynamic model of computation, and simulation phases can be considered similarly.
3) The same rules apply to combining simulators themselves, as any other tag system.
4) Simulation Phases additionally have “collaborative” semantics not associated with model time points

Effectively the model of computation, used by one simulator for it’s phases, can be combined with those of other simulators; and this is in addition to the combination of the model(s) of computation offered by the different simulators for it’s models. The combined tag system that results we define as having ’State’.

Phases are typically “global” (within a simulator), and there is agreement about their semantics. Just as when combining models there must be agreement about the meaning of time points, so there must be agreement about semantics when combining “phases” to form a wider “State”. This will be discussed later in the paper.

The point that phases adhere to a dynamic model of computation is important. Today “phases” are typically statically scheduled by a central simulator. Moving to a dynamic notion is a key to federating simulation, but also may help in alleviating some of the issues with phases.

Finally, unlike model “tags”, an emerging property of phases is that they are collaboratively entered into and exited from, by all those participating. This notion of “collaboration” is missing from normal simulation semantics (and models of computation). In principle all those models within a simulator that need to be activated and process during a “Phase” would be allowed to do so, and the simulator would not move on to another “Phase” until all those models had completed their activity. This does not limit the models of computation: Phases (and the associated actions carried out by models and simulators) could potentially happen sequentially or in parallel, the limitation is only at the beginning and end of a Phase.

II. Its just a Phase

Invariably, as simulation environments mature, dependencies grow, and phases are added. Phases get introduced to co-ordinate more or less globally between the components of a simulation. As systems become more complex, phases are used to simplify the execution steps of a simulation environment, requiring everybody to agree on what should happen in each phase. Before long, in the life of a simulation environment, two requirements emerge first the ability to co-ordinate between components, and second to perform some global tasks. Often these both get conflated into the “Phases” solution. Having a global approach to triggering components to perform some sort of action is clearly appealing from a purely programming perspective. The result of an organic approach can be seen inside QEMU [2] - nobody seems to fully understand the phases and dependencies, as different parts were added at different times for different purposes, it may be time for change!

The (single) simulation environment typically dictates when phases happen, based on it’s knowledge of the state of all the components. As those simulation environments become heterogeneous, federating a number of simulation environments together, the issue becomes co-ordination.

The challenge is to define a good set of phases, and equally to select the correct phase for a specific use case; further, to do so without knowledge of the specific models that will be written. To take a concrete case: Imagine a device to be tested (DUT). Clearly, before running the test, the DUT needs to be instantiated; then some DUT elaboration actions like determining vector sizes may need to be performed; next some failure conditions may need to be set; subsequently some test infrastructure will need to be constructed; then invariably debug and tool connections will need to be established. The list goes on, constructing “phases” for each of these actions soon explodes.

In the ”young” SystemC [3] language, the defined phases were:

1) constructors
2) before_end_of_elaboration

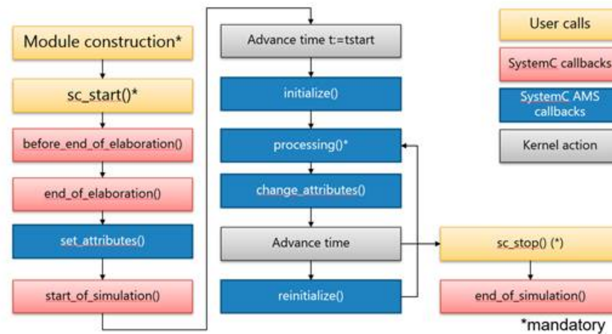


Figure 1: SystemC and SystemC AMS phases

3) end_of_elaboration
4) start_of_simulation
5) end_of_simulation

The class constructors chained together to build all the components and connections effectively form a phase, though we typically don't call it that. Phases 1 through 3 are preparation, 4 is execution, and 5 is termination.

Note how SystemC provides a boundary between module construction and simulation preparation at **"before_end_of_elaboration"**. In the context of SystemC this is often a critical border between when structure like modules, channels and ports can be created, while after this border the ports must be connected.

This goes wrong, of course, when we run out of **before_end_of_elaboration** phases, for instance when a model is written independently of it's test bench. This is especially the case if components require own elaboration/configuration action in dependency of context. An example could be a module with a vector port, whereby the vector size depends on module parameter. Thus the vector size to be calculated during elaboration and ideally propagated to the bound vector signal.

And yes, true to form, SystemC 3.0 has added a "suspended" phase with associated callbacks (which probably sits awkwardly in the "execution"/"termination" boundary).

Notice how these phases dont really say anything about the models being simulated, they are totally, selfishly, self-centred on the simulator and it's "state". Simulation phases are not likely to be empathetic, they would rather be dictatorial, verging on abusive. Models must adhere to SystemC's notion of "phases", SystemC does not adapt to the models requirements.

SystemC Analogue and Mixed Signal (AMS) [4] has its own callbacks for specific purposes. SystemC AMS first sets up the equation system or scheduling graph depending on the structure (e.g. an electrical network has modules for resistors, capacitors, etc which are connected by channels). During elaboration the structure is analysed and the equation system is set up, dynamically creating a (SystemC) process in which the SystemC AMS cluster will run. With the current SystemC version this is done in **end_of_elaboration** – previously this had to be done in **before_end_of_elaboration** in order to guarantee some dummy structures and work round the constraints of SystemC - this lead to chasing the 'last' **before_end_of_elaboration** phase.

The SystemC AMS phases must be integrated into the SystemC phases. This correlation between the phases is represented in figure 1:

Ideally, the SystemC AMS elaboration phases are executed after the corresponding SystemC phases, to ensure, that the SystemC phase call backs can be used to execute combined actions for SystemC and SystemC AMS.

Meanwhile, UVM [5] has the following set of phases (take a deep breath):

1) build
2) connect
3) end_of_elaboration
4) start_of_simulation
5) run
6) pre_reset
7) reset
8) post_reset
9) pre_configure
10) configure
11) post_configure
12) pre_main
13) main
14) post_main
15) pre_shutdown
16) shutdown
17) post_shutdown
18) extract
19) check
20) report
21) final

Phases 1 to 4 are preparation, phases 5 to 17 are execution, and phases 18 to 21 are termination. The preparation and termination phases are "function" phases, meaning they do not consume time. The execution phases are SystemVerilog tasks, meaning they can consume time. There is a nuance here that is not obvious. The **run_phase** operates concurrently with phases 6 to 17. You can step through phases 6 to 17 while **run_phase** is still live and operating.

You would be forgiven for asking, where is the kitchen sink phase, never fear, that's covered too as you can also have user defined phases! This is a Phase system with feelings!

It's unlikely that anybody uses all of these phases, but they are all probably used by somebody, somewhere.

For reference, and, again, to see the effect of time on a simulator original UVM phases, those that were in the first iteration of the package, were:

1) build
2) connect
3) end_of_elaboration
4) start_of_simulation
5) run
6) extract
7) check
8) report

The additional phases came in later when the committee decided to revisit the question. The IEEE standard for UVM contains the longer list of phases

Again, the role of the phases is to enforce ordering of operations and to ensure that dependencies are satisfied. If we had only one device, and thus no dependencies we wouldn't need phases. The only reason to have them is so that operations in phase x can be confident that all the operations in phase x-1 have already completed. For example, in the UVM connect phase you can rely on the fact that the components are all built so you don't end up trying to connect to a phantom component.

For most people, most of the time, the "classical" UVM phases are perfectly adequate. But then, people solve problems. Occasionally, putting things in the connect phases that are not connections because they need to run after the build is done, but before the simulation starts running. Or occasionally using
end_of_elaboration or **start_of_simulation** to deal testbench-specific ordering/dependency issues. Of course - there is only so far that game can take us. In the end, relying on a set of phases invariably will lead to "running out of phases".

As mentioned above, to handle the kitchen sink, and prevent the "running out of phases" nightmare, UVM has a facility for users to create their own phases and insert them into the phase graph. It even supports multiple independent phase graphs, called domains. The graphs have petri net semantics. When a phase terminates all of its children phases start executing. If a phase has multiple parents, it waits until all the parents have finished before it starts executing, creating a fork-join semantic.

But, when we get to the execution phases is when things get interesting. In SystemC, the devices run and that's it. They are responsible for taking care of their own internals – like reset or warm restart, etc. Maybe that's all that's necessary. Many UVM users are using older versions of UVM with the classic phases and don't seem to run into any issues they were unable to resolve within the UVM framework. But the newer UVM contains many "phases" that can happen during execution. One must ask the question, are these physical effects that should be simulated, or are they fascists of the simulation itself? Reset, for instance, would seem to be a real thing that should be "simulated" as any other model activity.

III. ONE SET OF PHASES TO RULE THEM ALL?

So, phases are painful, and different simulation environments have invariably ended up with "the kitchen sink" of phases. Every paper starts with the same cliche, things are becoming more complex. For modelling there is a trend towards larger systems of (heterogeneous) simulations being "federated" together. And thus the phase/state problem is born. How can an integrator handle the differences between phases, even if (as seems unlikely) they understand what all the different phases mean, and what the restrictions are. It would seem highly likely that there will be conflicts between them, or, at the very least, inabilities on behalf of one simulation environment or another to respond to some of those phases (especially where phases have been used as global trigger for a specific task).

Maybe this is all laziness. Maybe phases are an easy (and seemingly universal) way to order things during simulation, but, in the end always cause issues, leading to the "I've run out of phases" nightmare, at which point, new phases get added to simulators - rinse and repeat! Nobody ever agrees on what the Phases should actually be used for, so everybody wants their own Phases. Of course, if we could all agree on a specific set of phases, a united nation of phases, then the problem would be solved. But we don't even seem to be able to agree on a set of phases within a simulation environment - and we have seen how "running out of phases" just seems to be a ubiquitous problem in modelling. Without a clear understanding of what is guaranteed by each Phase, their use becomes difficult, increasing the pressure on phase expansion.

IV. PHASERS TO STUN!

In practice we have seen 3 approaches to phases. An ad-hoc collection of phases (e.g. within Qemu), a (more) constrained approach in which models must fit to the available phases (e.g. within SystemC), or the ability for models to define their own phases within a framework of existing phases (e.g. within UVM).

This extensible approach seems to push the problem to the integrator, who must then understand the phasing requirements of each individual component. However, another view of the same scenario would be that models that are dependent upon each other must know of each-other, and should be able to co-ordinate between themselves.

> **The role of the phases is to enforce ordering of operations and to ensure that dependencies are satisfied.**

Ensuring dependencies are satisfied and, indeed order is maintained, isn't always the prerogative of a central governing authority. Individuals can, if they agree to co-operate too ([6]). This shifts the problem to deciding and agreeing a set of ubiquitous global phases to more localised arrangements.

The question is whether a more localised framework is feasible, scales and allows models to be composed together. In essence, can the UVM notion of extensible phase be used universally to connect different environments. Or, put another way, can the simulation phases be treated as a model of computation in their own right?

This would 'complete' the spectrum of phase systems, from rigidly defined on the one hand to a complete dynamic system on the other.

Just as matter has more than 3 states ([7]), the expectation is that simulations will have many states, and that models and individual simulators may have multiple Phases ([8]). However,the suggestion made here is that additional phases need to be understood, the guarantees they make, and the ordering they require should be negotiated only between components, rather than treated globally. This is the notion of a "Dynamic" phase system.

To control a simulator, is to control how it advances through it's "Phases". To synchronise between simulators, or indeed models, requires some control over that progression. Hence the ability to hold a simulation in a Phase, neither executing, nor terminated, is critical.

Note that in almost all phases that we have seen, simulation time ceases to advance during a phase. The exception is a "run" phase itself, and the extensible phases in UVM. Hence it seems reasonable to use phases as a means to control the advance of simulation time.

There are multiple notions of time. Wall clock time always advances. Much as the authors have tried, there seems no way to turn back time [9] (if we neglect analog simulators which may permit a time rollback or iteration within a limited time interval). Simulation time, Local time, Quantum time, and other notions of time that simulators use, on the other hand, are in the control of the simulator and/or it's components.

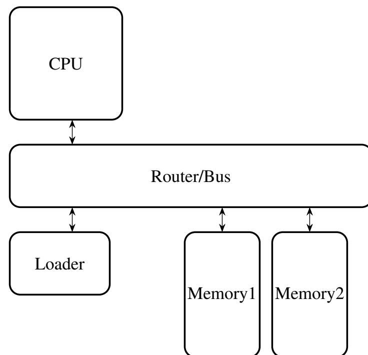
In general a phase could affect any of these "times" more or less locally, or, affect the advancement of simulation time globally.

- Suspend the simulation activity (stop time)
- Allow simulation to continue, but without time advancing
- Cause time to advance

A model could Demand, Request, Accept or Refuse to enter a phase, and hence cause or prevent the advancement of time. In this way, the movement between Phases is "collaborative".

Some examples may help to clarify what is being proposed as a 'dynamic' phased system:

Typically a memory is pre-loaded in simulation with a binary images, such that the CPU can execute that code. In SystemC this is often achieved during the **"end_of_elaboration"** phase, where models have been constructed and connected, but the simulation has not yet started.



In principle, the dependencies can be treated locally. The “loader” need only wait for its connection to the “router” to be established, and likewise for the “router” and “memory”. During this time, the router could consider itself in a “loading” phase. The “CPU” can start execution as soon as the “router” is satisfied that its “loading” phase has finished and permits the simulation to advance (a decision which is collaboratively arrived at by those models connected to the router). Hence the “router” could orchestrate this local part of the system. Note that the “loading” phase, which does not represent any simulate-able physical activity in the system, it is mealy there to enable simulation, would typically not allow the advance of simulation time.

While, in general, this seems feasible, it adds complexity. Further it relies upon being able to find connected devices, and discovering their status.

To take another example, other simulation environments have “phases” which are used for other “meta” configuration activity. For instance, one could imagine a “Special Debug Mode” configuration option, which would need to be transmitted to all the models in a system. This should be done prior to any other activity such that debug can be captured. To achieve this in a “dynamic” way, the simulation environment would require a means by which one element (model or tool) could “discover” others (in this case receptive to the “Special Debug Mode”).

Equally, it is possible to analyse a simulator such as SystemC in terms of the phases and the degree of collaboration it offers. For instance, while the initial simulation phases are effectively centrally orchestrated, (these could be viewed as a collection of phases that are agreed and subscribed to by all models in a SystemC simulation environment), the **“suspend”** phase is rather different. Individual models can request that the simulator enters a suspend phase (in which time will not advance). However, models may specifically enter an **“unsuspendable”** phase in which they deny the simulator from entering the suspend phase, and specifically, while in the **“unsuspendable”** phase, allow time to advance.

V. Dynamic Phasing

So maybe global simulation phases are the wrong thing to talk about. Maybe it’s better to focus, especially in today’s world of federated simulation, on individual components phases and how components interact with each other. The communication of phases, the means by which time is handled all require some degree of service from the simulation environment. One could imagine that this would cover publish and subscribe style services (which would be useful for the “Special Debug Mode” example), “Store and Query”, or indeed “Request and Reply”. Furthermore, much like in quantum physics, federated simulation interactions result in state entanglement, requiring each component to replicate its state across the network in an atomic, coherent, and deterministic manner. Any communication technology supporting this federation must address these inherent challenges of distributed processing. Difficult decisions, akin to Cornelian dilemmas, must be made to maintain strong consistency or, accepting eventual consistency in all circumstances [10]. These paradigms and challenges are typical handled in distributed data service systems, such as [11].

SystemC uses fixed phasing. The graph representing the schedule of phases is predetermined and it is up to the model builders to fit their model’s behaviour into the predefined set of phases. UVM also has a fixed set of phases. It also provides the capability to add new phases. Model builders can define whatever phases they wish with whatever semantics they wish for each new phase. However, the phase definitions must all be in place before run-time.

We believe that a fully dynamic phasing system would be preferable. In this system only an initial and final phase are defined before the simulation begins. As the simulation proceeds, each component tells the phasing executive which phase it wants to execute. It can provide some constraints, such as the new phase must execute before or after some other phase (or phases). The executive uses the information provided by the components to construct a phase schedule. It then executes the schedule. Execution of the schedule may cause components to add new phases to the schedule. The cycle continues until there are no more phases to execute.

VI. Conclusion

This paper tries to give a framework by which simulation environment can judge themselves, but those environments, and the components written in them will not change. This is merely theoretical.

Meanwhile the "federated phasing problem" is about connecting simulators together. The question of federation is pressing, and perhaps there are solutions.

Allowing for dynamic phases, and relying on simulation services to find and manage dependencies requires the cooperation of the different components. In this case, those components are, themselves, simulation environments. However as we are, industrially, in a state where few simulation environments interconnect, enforcing this approach now seems more achievable.

Federated simulations will be able to find other components on which they depend, especially those that they are connect to. Federation environments will have to ensure this. Collaboration between components that are aware of each other must also be possible, to organise specific "Phases" between themselves. Those may affect global simulation time. Phases may need to be swiftly transmitted locally, or broadcast globally.

Doing this seems achievable, has limited scope, and, perhaps, will provide a framework in which federation can be achieved!

References

[1] E. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, 1998.

[2] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41. Califor-nia, USA, 2005, p. 46.

[3] "Ieee standard for standard systemc® language reference manual," *IEEE Std 1666-2023 (Revision of IEEE Std 1666-2011)*, pp. 1–618, 2023.

[4] "Ieee standard for standard systemc(r) analog/mixed-signal extensions language reference manual," *IEEE Std 1666.1-2016*, pp. 1–236, 2016.

[5] "Ieee standard for universal verification methodology language reference manual," *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*, pp. 1–458, 2020.

[6] D. J. Dunn, "Articulating an alternative: the contribution of john burton," *Review of International Studies*, vol. 21, no. 2, p. 197–208, 1995.

[7] M. Wahab, *Solid State Physics Structure and Properties of Materials*. ALPHA Science International Limited (UK), 2005. [Online]. Available: https://books.google.co.uk/books?id=Cgb7ngEACAAJ

[8] A. Einstein, "Quantentheorie des einatomigen idealen Gases. (German) [Quantum theory of monatomic ideal gases]," vol. ??, no. ??, pp. 261–267, 1924, see part 2 [**?**]. In the two papers, Einstein extended Bose's work on monatomic gases [**?**], [**?**] to predict the Bose–Einstein effect (and likely, develop Bose–Einstein statistics).

[9] Cher, "If i could turn back time," https://en.wikipedia.org/wiki/If_I_Could_Turn_Back_Time, 1989.

[10] Z. Team, "Keeping storages aligned in Zenoh," https://zenoh.io/blog/2022-11-29-zenoh-alignment/, 2022, [Accessed 29-06-2024].

[11] Zenoh, "Zenoh-flow 0.6.0-rc: Getting started; zenoh - pub/sub, geo distributed storage, query - zenoh.io," https://zenoh.io/blog/2024-01-31-zenoh-flow-getting-started/, 2024, [Accessed 29-06-2024].