Verifying Concurrency in an Adaptive Ocean Circulation Model

Alper Altuntas
National Center for Atmospheric Research
Boulder, CO
altuntas@ucar.edu

John Baugh
North Carolina State University
Raleigh, NC
jwb@ncsu.edu

ABSTRACT

We present a model checking approach for verifying the correctness of concurrency in numerical models. The forms of concurrency we address are from (1) coupled modeling where distinct components, e.g., ocean, wave, and atmospheric, exchange interface conditions during runtime, and (2) multi-instance modeling where local variations of the same numerical model are executed concurrently to minimize common (and therefore redundant) computations. We present general guidelines for representing these forms of concurrency in an abstract verification model and then apply them to an adaptive ocean circulation model that determines the geographic extent and severity of coastal floods. The ocean model employs multi-instance concurrency: a collection of engineering design and failure scenarios are concurrently simulated using *patches*, regions of a grid that grow and shrink based on the hydrodynamic changes induced by each scenario. We show how concurrency inherent in the simulation model can be represented in a verification model to ensure correctness and to automatically generate safe synchronization arrangements.

CCS CONCEPTS

• **Software and its engineering** → **Model checking**; • **Mathematics of computing** → *Partial differential equations*; • **Theory of computation** → Parallel algorithms;

KEYWORDS

Scientific computing, finite element analysis, hurricane storm surge, concurrency, model checking

ACM Reference Format:
Alper Altuntas and John Baugh. 2017. Verifying Concurrency in an Adaptive Ocean Circulation Model. In *Proceedings of First International Workshop on Software Correctness for HPC Applications (Correctness 2017).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Predicting coastal floods from large-scale simulations of tropical storms is computationally demanding. Ocean circulation models, which attempt to capture the hydrodynamics as efficiently as possible, employ a variety of mechanisms to improve performance, including various forms of parallelism and adaptivity. Our study involves an adaptive multi-analysis code that eliminates redundant computations when multiple alternatives are being simulated, such as the topographic variations considered when designing protective structures and barriers. The approach, called adaptive subdomain modeling (ASM), concurrently analyzes any number of *child* domains, with each instance corresponding to a unique design or failure scenario, in addition to a full-scale *parent* domain providing the interface conditions at the boundaries.

A necessary condition for correctness is a safety property: a child domain without modifications must behave, hydrodynamically, like its parent. This condition has two key aspects. The first involves the scope of variables required to enforce interfaces on the child domains [5, 6], and the second concerns timing: when and how frequently to update their values within each iterative step of the simulation.

In this study we address the latter aspect, which brings concurrency into the picture because of possible race conditions on quantities that are copied from the parent to child domains along the interfaces of each child. During a concurrent run, a synchronization mechanism is to be employed to prevent the parent from overwriting pertinent data before they are retrieved by children, and to prevent children from using stale values of these quantities. Such guarantees are intended to be ensured by a *phasing mechanism* that regulates the progression of parent and child domains during each timestep.

Managing concurrency and ensuring that such guarantees hold in numerical models can be challenging. Conventional validation and verification methods like testing are both computationally demanding and incomplete. A complementary approach can in some cases be found in model checking, which is based on developing an abstract representation of a software system. A model checker interprets the abstract verification model, constructs a state space that includes arbitrary interleavings, and exhaustively searches it for any violations of the correctness properties of interest. In this study, we propose the use of such tools for verifying concurrent numerical models and present guidelines for a particular class of applications. We then develop an abstract verification model of concurrent ASM simulations in Promela, the input language of the SPIN model checker [10], to ensure correctness and to determine safe phase arrangements, i.e., the partitioning of timestep computations into phases.

In the remainder of the paper, we describe the types of concurrency we aim to address. We then outline the ASM technique, its basic computational features, and its realization in ADCIRC++ [4], our re-implementation of the popular ADCIRC ocean circulation model [14] with an updated software architecture. We then present our model checking approach, along with a verification model for concurrent ADCIRC++ simulations that we use to generate safe phase arrangements. We close with some perspectives on the approach and near-term directions for future research.

Correctness 2017, November 2017, Denver, CO USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 PROBLEM CONTEXT

Parallelism is often exploited to improve the computational performance of numerical models. It arises through domain decomposition and a variety of other avenues, including (1) coupled modeling and (2) multi-instance modeling. By coupled modeling, we mean the concurrent simulation of multiple phenomena, such as ocean, wave, and atmospheric models. By multi-instance modeling, we mean the concurrent simulation of multiple model instances, which might have local variations that represent parametric changes of interest to the modeler. Examples of coupled modeling include the combination of ADCIRC and SWAN [9] for surge and wave modeling, and global climate models like CESM [11]. Examples of multi-instance modeling include the data assimilation approaches used in climate modeling [16] and the ASM approach implemented in ADCIRC++, the focus of our study.

In the case of coupled modeling, one challenge is to determine the sequence of concurrent executions of components and the communication patterns among them, e.g., precipitation flux from atmospheric to land components, or fresh water flux from river to ocean components. If improperly handled or synchronized, the execution of these concurrent components may cause flux information or other updates to be lost before they are received by their counterparts.

Similar race conditions can arise in multi-instance modeling as well, if any communication is to be carried out between concurrent instances. In ASM, for instance, a parent domain must be prevented from overwriting quantities required by its children, and children must be prevented from overtaking a parent in a timestep. Beyond safety, we must also ensure that the synchronization mechanism is deadlock-free for any possible execution sequence and model configuration.

2.1 ASM

The ASM technique is based on concurrently executing a parent domain and a number of child domains with local modifications to eliminate redundant computations. Initially encompassing only the modified regions of a domain, a *patch*—the geographic extent of a child domain—grows and shrinks depending on the response of the model so as to contain the altered hydrodynamics originating from the modification. Figure 1 shows, in light blue, a small part of a parent domain, along with a darker green region associated with a child domain that evolves as the effects of a change propagate and eventually stabilize. The parent domain, shown in Figure 2, includes the western North Atlantic Ocean, the Caribbean Sea, and the Gulf of Mexico. An error indicator—a measure of the difference between the solutions in the parent and child domains—is calculated proximal to the boundaries of the patch to assess the spatial extent of the altered hydrodynamics. Results of our case studies show that the technique substantially reduces computational effort while maintaining accuracy [4].

The technique is implemented in ADCIRC++, a prototype based on the ADCIRC ocean circulation model [14] used by the U.S. Army Corps of Engineers (USACE), Federal Emergency Management Agency (FEMA), and others. Developed with data abstraction in mind, its updated software architecture is designed to facilitate adaptive behavior and to exploit concurrency both between and within domains. The current implementation employs shared memory parallelism for both of those types of concurrency.

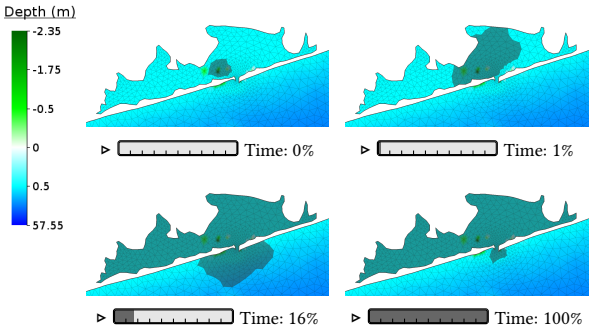

Figure 1: Expansion and contraction of a locally modified Shinnecock Inlet child domain patch at various timesteps.

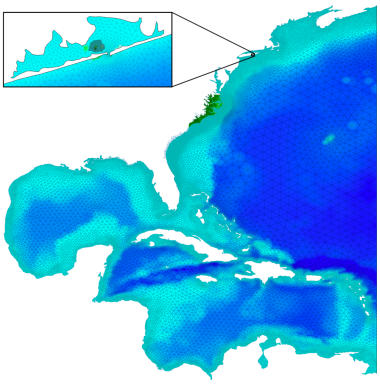

Figure 2: The region of interest shown on a parent domain that encompasses the western North Atlantic Ocean, the Caribbean Sea, and the Gulf of Mexico.

2.2 ADCIRC and ADCIRC++

The ADCIRC ocean model is based on a combination of finite element and finite difference techniques, which discretize time and space to solve systems of partial differential equations, and which operate on a grid of irregularly spaced nodes in a network of nonoverlapping triangular elements that represent the surface of the earth. Simulated time marches forward a step at a time, producing a history of basic nodal quantities, which include water surface elevations, velocities, and, to allow for advancing and receding flood waters, a wet/dry status indicating whether a node is considered "wet."

During an ADCIRC timestep, the following routines are executed in sequence: (1) timestep initialization to compute meteorological forcing, tidal forcing, ice effects, etc., (2) generalized wave continuity equation (GWCE) assembly to produce the coefficient matrix of the discretized GWCE, (3) GWCE solution based on a Jacobi conjugate gradient (JCG) sparse matrix solver that computes new water surface elevations, (4) determination of wetting and drying

status, and (5) momentum equation solution to compute updated velocities.

The original ADCIRC code is based on global, non-reentrant data structures and procedural decomposition, which complicate the implementation of ASM, since the approach is both adaptive and concurrent. Consequently, we designed a new software architecture and implementation in modern C++ (using the C++14 standard) based on object oriented design principles and data abstraction to facilitate adaptive behavior and to utilize concurrent and hierarchical executions of multiple domain instances.

The software architecture of the updated model, ADCIRC++, is implemented in two levels of source code. The first level, called OpenHDM, is a collection of abstract classes and class templates that form the general data architecture and provide methods for concurrent and hierarchical execution of multiple domain instances and the phasing mechanism. Its generic, dynamic, and reentrant data containers fully support adaptive grid behavior. For storing discrete model data, for instance, a `Grid`/`Patch` pair is implemented where the former is the container of the actual data e.g., nodes, cells, elements, etc., that are resized relatively less frequently at runtime, and the latter is a virtual view of the data, providing a level of indirection and allowing the designation of active regions of the grid that vary dynamically [1]. OpenHDM is available from an online repository [2]. The second level, consisting of concrete classes derived from OpenHDM, is the model-specific source code where the computational aspects of any model to be developed are implemented.

Similar to the software architecture itself, parallelism is implemented in two levels. The first level is inter-domain parallelism for the concurrent execution of multiple domain instances, and the second level is intra-domain parallelism based on decomposing a single domain instance into submeshes, each of which is executed by a dedicated core to reduce wall-clock time. While the implementation of inter-domain parallelism is provided in OpenHDM, intra-domain parallelism is implemented in the second level, the model-specific source code.

Our current prototype employs shared memory parallelism for both levels of parallelism. Doing so reduces coding time and eases experimentation with the new technique, but also increases complexity, especially at the inter-domain level, due to race conditions introduced by concurrent domains. We make use of the C++14 standard threading library for inter-domain parallelism because it fully supports object oriented design and data abstraction. As for intra-domain parallelism, we use OpenMP, which accommodates incremental parallelization [8]: our first approach was to simply parallelize nodal and elemental loops in timestep computations. As a next step, we implemented a domain decomposition approach based on dividing the computational mesh into multiple patches. Finally, we implemented an improved domain decomposition approach where we partitioned domain instances into multiple grids, thereby helping us resolve false sharing and improve memory locality [18]. Each incremental step required relatively modest effort, but we observed considerable improvement in scalability in the process.

2.3 Phasing mechanism

A key component of domain concurrency in ADCIRC++ is a synchronization mechanism, called *phasing*, implemented in OpenHDM. Based on grouping the computational routines (that constitute a timestep) into phases by taking into account the quantities that are transferred from parent to child domains, the mechanism prevents the parent domain from overwriting a quantity before it is received by the child domains, and prevents the child domains from moving ahead of their parents.

After having completed a phase, each domain evaluates several criteria before entering the next phase. For a child domain, the criterion is that its parent must have completed the phase it is about to enter. For a parent, the criterion depends on whether the phase to be entered ($p + 1$) is designated to be *concurrent* or not, which is decided when the phasing configuration is established. If phase ($p + 1$) is flagged as concurrent, i.e., if it can be executed by the parent while children are executing the previous phase (p), the criterion is simply that all children must have *entered* the phase that has just been completed by the parent (p). Otherwise, the criterion for the parent is that all children must have *completed* the previous phase (p). Figure 3 illustrates these criteria.

Note that, for flexibility in scheduling, one would prefer that all phases be designated concurrent. However, data dependencies—and other relationships affecting the way quantities are processed in each timestep—may preclude one from doing so.

In the case of ADCIRC++ and ASM, the computational routines to be grouped into phases are: (1) the timestep initialization, (2) GWCE assembly, (3) GWCE Solver, (4,5) the first and second half of the wet/dry algorithm, and (6) the momentum equations solver. The quantities to be copied from parent to children are (1) water surface elevations, (2) velocities, (3) wet/dry states, and (4) wind forcing parameters.

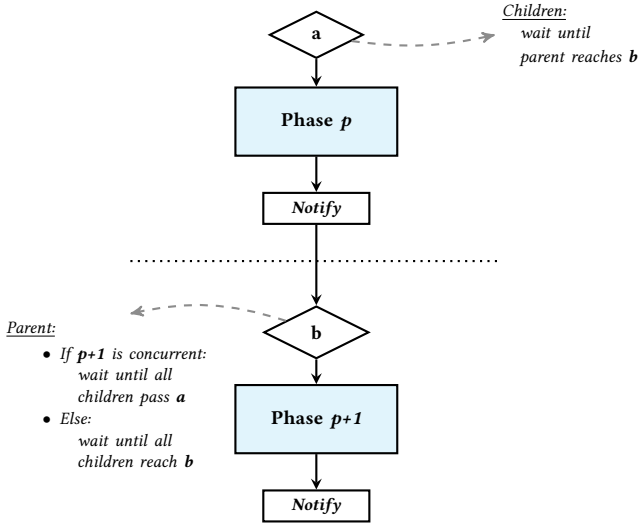

Figure 3: Phase execution criteria for parents and children.

3 MODEL CHECKING CONCURRENT NUMERICAL MODELS

Before introducing basic elements of our approach for modeling and verifying concurrent systems, we note the following terms used in our descriptions:

component: a concurrent and distinct model in a coupled system, e.g., ocean, atmosphere, ice, and land components in a climate modeling system.

instance (or domain): an original or locally modified model to be simulated, e.g., a parent or any of its child domains.

critical quantity: a physical value or quantity to be transferred from one component or instance to another.

Our verification approach aims to incorporate the following three constituents of a concurrent system in the abstract verification model:

(1) critical quantities
(2) concurrent components and instances
(3) synchronization mechanism (or a driver)

Basic elements incorporating these constituents in a verification model are as follows.

(1) *Critical quantities.* While critical quantities, e.g., masses, velocities, and fluxes, are typically represented by real variables in an actual numerical model, they can often be taken as discrete for purposes of verification. For instance, instead of explicitly representing actual computed values of such quantities, in the abstract model they denote the version or state of the actual quantity as an *integer* value, akin to the timestamp associated with an update. This abstraction approach is similar to data-type reduction, a method commonly used in model checking to reduce a potentially infinite range of values to a tractable, finite range [13, 15]. Additionally, we use only a single variable to represent a critical quantity for an entire computational grid. This abstraction offers substantial flexibility at a coarse level: a state change in the abstract critical quantity may represent a state change on a single grid node, on a specific set of nodes, or on the entire grid of the actual numerical model. Thus, we make no distinction regardless of the spatial extent of the state change to a critical quantity. Further, we treat uninterrupted state changes on a critical quantity over a subset or the entirety of a grid as a unified and atomic state change in the verification model. In other words, if a critical quantity is updated over an entire computational grid atomically, for instance, the abstract variable representing the critical quantity is updated only once.

In keeping with this view, we model just two abstract operations on the integer values associated with critical quantities: `write` and `copy`. The `write` (or update) operation, which increments its value by one, denotes a component or instance computing and updating the value of a critical variable. The `copy` operation, on the other hand, denotes a component or instance retrieving the value of the critical quantity from another concurrent component. Since the values of critical quantities in a verification model are rather a representation of the versions of the actual critical quantities, the `copy` operation in a verification model acts as a placeholder for correctness checks, where the modeler can specify safety properties, such as whether the correct versions of a critical quantity are always obtained.

(2) *Concurrent components and instances.* We represent each component or instance as a separate process that runs concurrently with any others. Doing so enables the model checker to take arbitrary interleavings and non-deterministic interactions into account. Ideally, the only properties to be incorporated in the abstract model would be those directly related to synchronization and communication aspects; limiting them as such further reduces the state space of the verification model.

(3) *Synchronization mechanism (or a driver).* Depending on the model checking tool to be utilized, synchronization constructs that regulate the execution of concurrent components or instances are incorporated in the verification model. The SPIN model checker [10], for instance, provides modelers with constructs and features including message channels, atomic blocks, multiple processes, and an expressive executability rule to effectively represent the communication and synchronization primitives necessary for concurrent components.

4 MODELING CONCURRENT ASM SIMULATIONS IN PROMELA

To verify the phasing mechanism, which is necessary for the correctness of the ASM technique, we now model ADCIRC++'s concurrent timestepping routine in Promela, a C-like modeling language that is interpreted by the SPIN model checker. We present the complete verification model in an online repository, where we also provide a detailed description of the model [3]. Here, we briefly describe key points and show how the approach outlined in the previous section can be applied in the context of an adaptive ocean model.

4.1 Critical quantities

The movement of data in ADCIRC++ between a parent and its children consists of basic quantities in ocean modeling: water surface elevations, velocities, and wet/dry states along the child domain interfaces, and wind forcing parameters within child domains. In addition to the abstractions outlined in Section 3 for critical quantities, we now employ a further simplification: instead of defining separate abstract variables for both a parent and its child domain, we define a single, global integer variable that represents the states of both domains for each critical quantity.

As suggested, the parent increments these global variables to indicate an update in its critical quantities. The child, however, now decrements these abstract global variables to indicate an update in its own critical quantities. In effect, these abstract global variables now keep track of the "version difference" between the states of parent and child domain instances for a given critical quantity. Therefore, the value of the abstract global variable being zero before a copy operation indicates that the value being copied by the child is the same as the value the parent had when it was at the same location in a timestep. A value greater than zero, however, indicates that the value being copied was computed by the parent in a subsequent stage.

The only other operation on critical quantities is `copy`, which is also represented in an abstract manner. Instead of copying data, the abstract model incorporates a safety check to determine whether the copy operation could ever be unsafe, i.e., whether an incorrect

version of a critical quantity could be copied in any potential execution sequence. This safety property is specified using a linear temporal logic (LTL) property, whose details are presented in the online repository of the verification model [3].

4.2 Concurrent parent and child domain instances

In a typical ASM project, as many as 50 or 100 concurrent child domains may be defined, but here—without loss of generality—we model only a single child domain, along with a parent domain. We are able to do so in our case because data transfer is one-way from a parent to its children, and because child domain simulations are structurally designed to avoid interfering with each other. Such guarantees enable a form of compositional reasoning often leveraged in model checking [7, 15].

In an actual ADCIRC++ simulation, six computational routines are called in sequence at every timestep. These routines contain numerical and logical operations carried out over the entire computational grid to obtain a time history of nodal quantities. In the verification model, however, we abstract from the computational aspects and incorporate only the two operations `write` and `copy` on the critical quantities, as outlined.

We represent the concurrent execution of parent and child domains using two active Promela processes, one for each concurrent domain. These processes execute the `main` function shown in Algorithm 1, which calls the timestepping function infinitely many times. Because of this, and since the execution of concurrent processes is interleaved and non-deterministic in Promela, we simulate all possible concurrent execution sequences that can occur during an actual run.

As seen in the algorithm, `write` operations are carried out at the same locations within a timestep for both the parent and child domain processes. These operations correspond to computing and assigning the given quantities in the actual ADCIRC++ timestepping routines. The child domain process additionally carries out `copy` operations after its `write` operations. These correspond to retrieving data from the parent as a corrective step so that the parent's contributions are incorporated in the child domain at its interface with the parent. In other words, these steps are where data from the parent are used to enforce the boundary conditions of the child domain.

```
function exec_timestep()        // advances one step in time
    // Routine 0: Timestep Initialization
    write(τ)                          // compute and store winds
    if isChild() then
        copy(τ)                       // copy to update winds
        copy(η)                       // copy to assess hydrodynamics
        copy(υ)                       // copy to assess hydrodynamics
    // Routine 1: GWCE Assembly
    skip                              // no operations on critical quantities
    // Routine 2: GWCE Solver
    write(η)                          // compute and store elevations
    if isChild() then
        copy(η)                       // copy to update elevations
    // Routine 3: Wet/Dry Algm (1st half)
    write(η)                          // update elevations of dry nodes
    write(wd)                         // compute and store wet/dry states
    if isChild() then
        copy(wd)                      // copy to enforce wet/dry states
    // Routine 4: Wet/Dry Algm (2nd half)
    write(wd)                         // compute and store wet/dry states
    if isChild() then
        copy(wd)                      // copy to enforce wet/dry states
    // Routine 5: Momentum Equations
    write(υ)                          // compute and store velocities
    if isChild() then
        copy(υ)                       // copy to enforce velocities

function main()
    while True do
        call exec_timestep()
```

Algorithm 1: Abstract timestepping loop for parent and child domains, where τ, η, υ, and wd are wind forcing parameters, water surface elevations, velocities, and wet/dry states, respectively. This abstract model includes only the operations on critical quantities.

4.3 Phasing mechanism

Before instantiating the concurrent parent and child domain processes, the verification model non-deterministically arranges the six timestep routines by grouping them into phases where each phase contains one or more consecutive routines. It also decides non-deterministically whether a phase is concurrent, i.e., whether it can be executed concurrently by the parent while the child is executing the previous phase. If a phase is not concurrent, then the parent waits for its children to catch up before entering into that phase.

Once the phases are determined, the timestepping processes of the parent and child are instantiated. During the execution of the timestepping routines, the consecutive phases of these domain instances are concurrently executed. Before entering into a new phase, both domains evaluate the criteria described in Section 2.3. In the actual ADCIRC++ code, we make use of synchronization primitives from the standard threading library of C++14, including `std::mutexes`, `std::condition_variables` and various types of locks, to implement these criteria. In the verification model, however, we make use of SPIN's executability rule to replicate the wait-notify behavior: a statement can be executed only if it evaluates to true. Thus, the statements corresponding to the criteria for entering into a new phase blocks the processes of the parent and child until they become true, which may happen when the other domain changes the state of the model.

The soundness of the phasing mechanism depends not only on the criteria regulating the progression of concurrent parent and

child domains from one phase to another, but also on the arrangement of the computational routines as phases. The timestep routines must be grouped into phases in such a way that all potential race conditions on the critical quantities are eliminated. As an example, Figure 4 illustrates an unsafe concurrent execution sequence due to the following faulty phase arrangement randomly generated by SPIN:

- **Phase 0:** Routines 0 and 1. (*not concurrent*)
- **Phase 1:** Routines 2 and 3. (*concurrent*)
- **Phase 2:** Routines 4 and 5. (*not concurrent*)

The concurrent execution timeline of the parent and child domains (Figure 4) reveal that, during the last copy operation shown, `copy(`η`)`, the value of the abstract critical quantity to be copied is not zero. This erroneous state in the verification model represents the case where the value copied by the child is not the same as the value that the parent had when it was at the same location within that phase (Phase 1). What the child gets from the parent, instead, is an overwritten value. The parent overwrites this quantity at $t = 4$. What the child should have received, however, is the value that the parent had at $t = 3$.

The defect in the phase arrangement leading to this safety violation is due to the fact that two separate write operations on the same critical quantity (η) are placed in the same phase (Phase 1) which illustrates the danger of the parent overwriting the first computed value of the quantity before it is copied by the child. Note that this is not the only type of defect that may exist in a phasing arrangement. Notice, for instance, that a violation could have occurred earlier in the timeline had the first `write(`η`)` operation carried out by the parent at $t = 3$ occurred before the `copy(`η`)` operation carried out by the child. This error would result from a phase arrangement defect where a concurrent phase includes a write operation on a critical quantity to be copied by the child at the preceding phase.

4.4 Verification approach

Having modeled concurrent ADCIRC++ simulations in Promela, we utilize the SPIN model checker to verify that the phasing mechanism ensures the correctness of concurrency and data movement in ASM simulations, provided that a safe phase arrangement is employed. Thus, the verification effort involves determining those safe phase arrangements.

The first property we consider is an assertion stating that the abstract value of a critical quantity during a copy operation must always be non-negative. The interpretation of this property is that a child domain should never be able to overtake its parent and copy a critical quantity. The model checker does not find any violation of this property for any possible phasing arrangement, confirming that the mechanism is able to prevent this undesired behavior regardless of the choice of phase arrangement.

Our next safety property states that a parent can never overwrite a critical quantity before it is copied by the child. This property is similarly expressed as an assertion statement evaluating the value of the abstract critical quantity to be copied. The SPIN analysis of this safety property indicates counterexamples pointing to the occurrence of the undesired behavior where the parent advances more than it should and overwrites a critical quantity.

To determine the safe phasing arrangements, i.e., the ones that prevent this undesired behavior, we specify an LTL property which states that all phase arrangements must eventually lead to an execution sequence in which the child copies an incorrect version of a critical quantity. Since SPIN tries to find a counterexample for a given property, the existence of an *error* for this correctness property actually reveals the phase arrangements where the versions of the copied critical quantities can never be incorrect, i.e., where the child domain always receives the correct versions of the critical quantities. Recall that the undesired behavior in question occurs when the parent advances too far during a timestep. Thus, we administer a stress test by setting the priority of the parent process to be higher than that of the child process.

For a maximum of 6 phases, the SPIN model checker finds 324 counterexamples. Since these counterexamples correspond to the execution sequences where the child can never copy an incorrect version of a critical quantity, the phase arrangements in these counterexamples correspond to safe ones that prevent unsafe advancement of domains and incorrect data movement between them. Using a simple Python script (again, provided online [3]), we extract all of the unique safe phase arrangements, which are 108 in total. Taking the number of phases and the rate of concurrency into account, we adopt the following phase arrangement in ADCIRC++:

- **Phase 0:** Routine 0. (*concurrent*)
- **Phase 1:** Routine 1. (*concurrent*)
- **Phase 2:** Routine 2. (*concurrent*)
- **Phase 3:** Routine 3. (*not concurrent*)
- **Phase 4:** Routines 4 and 5.(*concurrent*)

Note that Phase 3, which includes the first half of the wetting and drying algorithm, is determined to be "not concurrent" for safety. This result is due to water surface elevations, which are copied by the child domain in Phase 2 and modified in both Phase 2 and Phase 3, so the parent waits for children to finish Phase 2 before it starts Phase 3. This prevents the parent from overwriting the elevations in Phase 3, before the children copy this critical quantity in Phase 2.

5 CONCLUSIONS

The concurrency inherent in coupled and multi-instance modeling creates both opportunities and challenges that may benefit from static analysis. In this study we show how model checking in SPIN can be used to find race-free phasing mechanisms in an adaptive ocean circulation model. Our coarse-grained approach requires only relatively modest levels of human and computer effort: the initial verification approach was put together in less than a day, and the final model consists of just under 200 lines of Promela code. As for computational efficiency, the SPIN model checker takes only a couple of seconds on a laptop computer to generate all safe phase configurations, whereas typical ADCIRC++ test runs require thousands of CPU hours.

Compared to mechanical methods like automated model extraction, relying on manual translation of code into Promela—as we have proposed—means we might miss lower-level defects or even introduce errors that invalidate our verification approach. As with any other modeling activity, however, steps can be taken to gain confidence in a model by exercising it and by comparing it against

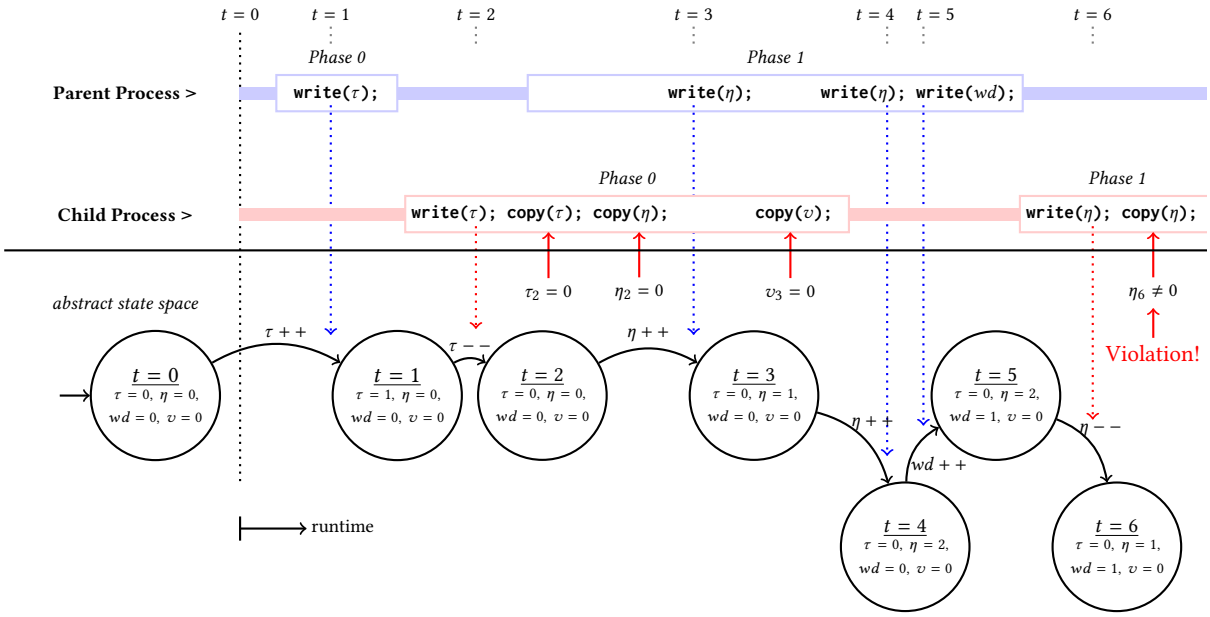

Figure 4: The timeline of an unsafe concurrent execution sequence due to a faulty phase arrangement. Increments and decrements in critical quantities correspond to write operations by the parent and child, respectively. A copy operation is safe only if the value is zero at the start of a copy operation.

reality. The perspective we adopt is the one employed by advocates of lightweight formal methods [12], where model checkers and other tools are judiciously directed at trickier issues that may not lend themselves to informal reasoning. Another example of this lightweight approach can be found in a verification study [5] involving ADCIRC's wetting and drying algorithm in Alloy.

We have found that model checking, though infrequently encountered in scientific computing [5, 17], helps in managing the complexities associated with concurrency in large-scale numerical programs. In addition to helping ensure sound implementations, the outlined approach may also be used in the context of performance optimization. As a future direction, for instance, we plan to utilize the approach for a coupled model to generate a set of safe concurrency configurations (defined by component sequencing and communication patterns) to be considered as a *search space* where a cost function is defined and solved to obtain optimal concurrency configurations.

REFERENCES

[1] Alper Altuntas. 2016. *An Adaptive Multi-Analysis Technique and Software Architecture for Ocean Circulation Models.* Ph.D. Dissertation. North Carolina State University.

[2] Alper Altuntas. 2016. OpenHDM. https://github.com/alperaltuntas/OpenHDM. (2016).

[3] Alper Altuntas. 2017. Promela Models of ASM Phasing Mechanism. https://github.com/alperaltuntas/verifyPhasing. (2017).

[4] Alper Altuntas and John Baugh. 2017. Adaptive subdomain modeling: A multi-analysis technique for ocean circulation models. *Ocean Modelling* 115 (2017), 86–104.

[5] John Baugh and Alper Altuntas. 2016. Modeling a Discrete Wet-Dry Algorithm for Hurricane Storm Surge in Alloy. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings.* Springer International Publishing, Cham, 256–261.

[6] John Baugh, Alper Altuntas, Tristan Dyer, and Jason Simon. 2015. An exact reanalysis technique for storm surge and tides in a geographic region of interest. *Coastal Engineering* 97 (2015), 60–77.

[7] Sergey Berezin, Sérgio Campos, and Edmund M Clarke. 1998. Compositional reasoning in model checking. *Lecture Notes in Computer Science* 1536 (1998), 81–102.

[8] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

[9] JC Dietrich, M Zijlema, JJ Westerink, LH Holthuijsen, C Dawson, Richard A Luettich, RE Jensen, JM Smith, GS Stelling, and GW Stone. 2011. Modeling hurricane waves and storm surge using integrally-coupled, scalable computations. *Coastal Engineering* 58, 1 (2011), 45–65.

[10] Gerard Holzmann. 2003. *Spin model checker, the: primer and reference manual.* Addison-Wesley Professional.

[11] James W Hurrell, Marika M Holland, Peter R Gent, Steven Ghan, Jennifer E Kay, Paul J Kushner, J-F Lamarque, William G Large, D Lawrence, Keith Lindsay, et al. 2013. The community earth system model: a framework for collaborative research. *Bulletin of the American Meteorological Society* 94, 9 (2013), 1339–1360.

[12] Daniel Jackson and Jeannette Wing. 1996. Lightweight formal methods. *IEEE Computer* 29 (1996), 22–23.

[13] Ryan Kirwan, Alice Miller, Bernd Porr, and P Di Prodi. 2013. Formal modeling of robot behavior with learning. *Neural computation* 25, 11 (2013), 2976–3019.

[14] Richard Albert Luettich and Joannes J Westerink. 2004. *Formulation and numerical implementation of the 2D/3D ADCIRC finite element model version 44. XX.* R. Luettich.

[15] Kenneth McMillan. 1999. Verification of infinite state systems by compositional model checking. *Correct Hardware Design and Verification Methods* (1999), 705–705.

[16] Kevin Raeder, Jeffrey L Anderson, Nancy Collins, Timothy J Hoar, Jennifer E Kay, Peter H Lauritzen, and Robert Pincus. 2012. DART/CAM: An ensemble data assimilation system for CESM atmospheric models. *Journal of Climate* 25, 18 (2012), 6304–6317.

[17] Stephen F Siegel, Anastasia Mironova, George S Avrunin, and Lori A Clarke. 2008. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17, 2 (2008), 10.

[18] Christian Terboven, Dirk Schmidl, Henry Jin, Thomas Reichstein, et al. 2008. Data and thread affinity in openmp programs. In *Proceedings of the 2008 workshop on Memory access on future processors: a solved problem?* ACM, 377–384.