ALTUNTAS, ALPER. An Adaptive Multi-Analysis Technique and Software Architecture for Ocean Circulation Models. (Under the direction of Dr. John Baugh.)

Hydrodynamic models are widely used by scientists and engineers to assess the impacts of coastal hazards such as tides, hurricane storm surge, and wind waves. These models require a substantial amount of computational resources due to the geographical extent of coastal processes and complex nature of ocean physics. When multiple what-if scenarios are to be evaluated, either as part of a forecasting/hindcasting study or a sensitivity analysis, the computational demand further increases.

This dissertation presents an adaptive multi-analysis technique to increase the computational efficiency of ocean models when multiple local scenarios are to be evaluated. The technique, called adaptive subdomain modeling, is realized by the concurrent execution of multiple child domains, each corresponding to a local scenario, and a full-scale parent domain providing the boundary conditions for the child domains. During runtime, the spatial extent of each child domain is adaptively adjusted depending on the movement of altered hydrodynamics, i.e., the differences between the solutions of the child domain and the parent domain, to maintain both the efficiency and the reliability.

In addition to the adaptive subdomain modeling approach, the study includes the development of a generic software architecture for numerical ocean models, based on object oriented design principles and data abstraction. The new architecture utilizes concurrent executions of multiple domain instances and facilitates adaptive behavior. To demonstrate the effectiveness and applicability of the software architecture, we develop a simple finite-volume shallow water model and re-implement ADCIRC, an advanced finite-element ocean circulation model, with the incorporation of the adaptive subdomain modeling approach.

While object-oriented design principles reduce the complexity of numerical models and enhance their ability to accommodate new computational techniques, verification of the algorithm-
mic behavior of model components with the inclusion of new features, like adaptive subdomain modeling, still remains a challenge. Therefore, this study proposes the application of model checking tools for the formal verification of the algorithmic properties of existing and newly incorporated model components. As an example application, a generalized abstraction and verification modeling guideline is provided for the formal verification of wetting and drying algorithms of ocean models. Subsequently, the correctness of ADCIRC’s wetting and drying algorithm for the child domains in the adaptive subdomain modeling approach is verified using SPIN model checker.
An Adaptive Multi-Analysis Technique and Software Architecture for Ocean Circulation Models

by

Alper Altuntas

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Civil Engineering

Raleigh, North Carolina
2016

APPROVED BY:

Dr. Joel Dietrich

Dr. Gnanamanikam Mahinthakumar

Dr. Alina Chertock

Dr. John Baugh
Chair of Advisory Committee
DEDICATION

To my family.
BIOGRAPHY

The author was born in 1989 in Giresun, Turkey. He completed his undergraduate degree in civil engineering at Istanbul Technical University in 2010, and joined the Master of Science program in civil engineering at North Carolina State University. After completing his master's study in 2012, he pursued his doctoral study at NCSU.
ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. John Baugh, for his support and guidance throughout my study at NCSU. It has been a great privilege and experience to work with him. I would also like to thank my committee members, Dr. Casey Dietrich, Dr. Mahinthakumar, and Dr. Alina Chertock for their helpful feedback and insight. Many thanks to Tristan Dyer, Julie Rutledge, and Jason Simon for their contributions to subdomain modeling.
# TABLE OF CONTENTS

LIST OF TABLES .............................................................. vii

LIST OF FIGURES ............................................................ viii

Chapter 1 Introduction ...................................................... 1

Chapter 2 Adaptive Subdomain Modeling: A Multi-Analysis Technique for Ocean Circulation Models ................................................. 5
  2.1 Introduction ............................................................ 6
  2.2 Background ............................................................ 8
    2.2.1 ADCIRC ......................................................... 8
    2.2.2 Conventional subdomain modeling and applications .......... 9
  2.3 Adaptive subdomain modeling ....................................... 12
    2.3.1 Error indicator ............................................... 12
    2.3.2 Adaptivity algorithm ......................................... 15
    2.3.3 Boundary conditions ......................................... 26
  2.4 Workflow and hybrid approach ..................................... 27
  2.5 Test cases ................................................................ 28
    2.5.1 ASM control parameters ...................................... 29
    2.5.2 Applications and Performance ................................. 49
  2.6 Conclusions and future work ....................................... 54

Chapter 3 A Generic Software Architecture for Concurrent and Adaptive Ocean Models .............................................................. 57
  3.1 Introduction ............................................................ 57
  3.2 Background ............................................................ 59
    3.2.1 Data Abstraction and OOP .................................... 59
    3.2.2 ASMFV Formulation ............................................ 61
    3.2.3 ADCIRC Background ........................................... 63
    3.2.4 Adaptive Subdomain Modeling ................................ 64
  3.3 Related Work .......................................................... 66
  3.4 The Architecture ....................................................... 68
    3.4.1 OpenHDM ....................................................... 69
    3.4.2 Inter-Domain Concurrency .................................... 74
  3.5 ASMFV .................................................................... 77
  3.6 ADCIRC++ ................................................................ 79
    3.6.1 Derived classes of ADCIRC++ ................................. 80
    3.6.2 Application of ASM in ADCIRC++ .......................... 83
    3.6.3 A case study: Hurricane Irene (2011) storm surge .......... 84
    3.6.4 Discussion ......................................................... 85
  3.7 Conclusions .............................................................. 87
Chapter 4 Model Checking Wetting and Drying Algorithms .............................................. 89
  4.1 Introduction .................................................. 89
  4.2 Motivation ..................................................... 92
  4.3 Background ................................................... 94
    4.3.1 ADCIRC Background ..................................... 94
    4.3.2 Subdomain Modeling ..................................... 96
    4.3.3 SPIN and Promela ......................................... 99
  4.4 Abstracting Wetting and Drying Algorithms .................................................... 106
    4.4.1 Discretization in Space and Time ......................... 106
    4.4.2 Data and Control Flow .................................. 108
  4.5 Case Study: ADCIRC’s wetting and drying algorithm ........................................ 109
    4.5.1 Abstract Grid .............................................. 110
    4.5.2 Control Flow .............................................. 112
    4.5.3 Elemental drying with three wet nodes ................. 117
    4.5.4 Verification of wetting and drying in subdomains ..... 119
  4.6 Limitations .................................................. 124
  4.7 Related Work ................................................ 128
  4.8 Conclusions .................................................. 128

BIBLIOGRAPHY .......................................................... 130

APPENDICES ............................................................ 142
  Appendix A ADCIRC++ Input File Formats .................................................. 143
  Appendix B Auxiliary inline and macro definitions .................................. 145
## LIST OF TABLES

Table 2.1  Comparison of computational costs for Case 5.  
Table 2.2  Comparison of computational costs for Case 6.  

Table A.1  Format of ADCIRC++ project input file, where **header** is an alphanumeric description of the input file, **projectID** is an alphanumeric id for the project, **nd** is the number of domains, **domainID**

Table A.2  Format of ADCIRC++ domain configuration file, where **header** is an alphanumeric description of the input file, **domainType** is the type of the domain (="ADCIRC" for ADCIRC domains), **type** is the type of the i.th input file (=14 for grid files, =15 for model configuration files, etc.), **format** is the format of the i.th input file (="ADCIRC" for ADCIRC domains), **fileDir** is the directory of the i.th input file relative to the directory of the configuration file. 

Table A.3  Format of ADCIRC++ .dif file, where **np** is the number of properties to be locally modified, **propertyID** is the id of i.th property to be modified (e.g., "bathymetry", "mannings_n_at_sea_floor", etc.), **nn** is the number of nodes with modified propertyID, **ni,j** and **vali,j** are the node number and the new value of the modified property i of the j.th node in the set of nodes with modified propertyID. 

Table A.4  Format of optional ADCIRC++ .asm file, where **header** is an alphanumeric description of the input file, **tau** is the tolerance, **sigma** is the safety factor, **lambda** is the decay constant, **theta** is the minimum activation interval.
LIST OF FIGURES

Figure 2.1 Expansion and contraction of a locally modified Shinnecock Inlet child domain patch at various timesteps. ......................................................... 8
Figure 2.2 Subdomain extraction with the SMT user interface. .......................... 11
Figure 2.3 Flowchart of the modified timestepping loop for adaptive subdomain modeling (ASM steps shaded, steps common to ASM and CSM patterned, and original steps left unshaded). .................................................. 13
Figure 2.4 Initial patch of the modified Shinnecock child domain. ..................... 16
Figure 2.5 Relationship between tolerance and error indicator during a simulation. The patch expands at timesteps $t_1$, $t_3$, and $t_4$, and contracts at timestep $t_2$ since $\rho_{\text{max}}$ falls beneath $\sigma \tau^0$. .................................................. 17
Figure 2.6 Summary of criteria for expanding a patch. .................................... 19
Figure 2.7 Illustration of criteria for expanding a patch. .................................... 19
Figure 2.8 Summary of criteria for contracting a patch. ..................................... 20
Figure 2.9 Illustration of criteria for contracting a patch. ................................... 21
Figure 2.10 Expansion stage of the adaptivity algorithm. .................................... 22
Figure 2.11 Main steps of the expansion of a child domain patch, where the patch boundary node $r$ is marked for expansion at the first step of the algorithm. ....... 23
Figure 2.12 Contraction stage of the adaptivity algorithm. ................................. 24
Figure 2.13 Main steps of the contraction of a child domain patch, where the patch boundary node $r$ is marked for contraction at the first step of the algorithm. ... 25
Figure 2.14 The complete workflow of a typical ADCIRC++ run with ASM. ............ 28
Figure 2.15 Case 1: Local change near Shinnecock Inlet. ................................. 30
Figure 2.16 Case 1: Progression of a Shinnecock child domain patch for $\tau^0 = 10^{-3}$, $\sigma = 10^{-2}$, $\lambda = 10^{-4}$, and $\theta = 10$, with elements in the patch darkened. Snapshots: (a) initial extent, (b) expansion at timestep 812, (c) largest patch (occurring at timestep 2949), and (d) final extent (timestep 71276). ......................... 31
Figure 2.17 Case 1: $l^2$-norms of maximum elevation errors in Shinnecock child domains. 32
Figure 2.18 Case 1: max-norms of maximum elevation errors in Shinnecock child domains. 33
Figure 2.19 Case 1: Average percentage of active elements in Shinnecock child domains (with $\lambda$, $\tau$ axes reversed). .............................................. 34
Figure 2.20 Case 2: Local change at Hatteras Inlet. ........................................ 35
Figure 2.21 Case 2: Progression of a Hatteras child domain patch for $\tau^0 = 10^{-3}$, $\sigma = 10^{-2}$, $\lambda = 5 \times 10^{-4}$, and $\theta = 100$, with elements in the patch darkened. Snapshots: (a) initial extent, (b) expansion at timestep 1905, (c) largest patch, and (d) final extent (timestep 691196). .............................................. 36
Figure 2.22 Case 2: $l^2$-norms of maximum elevation errors in Hatteras child domains. 36
Figure 2.23 Case 2: max-norms of maximum elevation errors in Hatteras child domains. 38
Figure 2.24 Case 2: Average percentage of active elements in Hatteras child domains. 38
Figure 2.25 Case 3: Local change at Walden Creek in the Cape Fear (Hurricane Fran, 1996) subdomain [12]. ......................................................... 39
Figure 2.26 Case 3: Progression of a Walden Creek child domain patch for $\tau^0 = 10^{-5}$, $\sigma = 10^{-2}$, $\lambda = 5 \times 10^{-4}$, and $\theta = 10$, with elements in the patch darkened. Snapshots: (a) initial extent, (b) extent at timestep 546,520, (c) largest extent (occurring at timestep 570,694), and (d) final extent (timestep 668,367).

Figure 2.27 Case 3: $l^2$-norms of maximum elevation errors in Walden Creek child domains.

Figure 2.28 Case 3: max-norms of maximum elevation errors in Walden Creek child domains.

Figure 2.29 Case 3: Average percentage of active elements in Walden Creek child domains (with $\lambda$, $\tau$ axes reversed).

Figure 2.30 Case 4: Local change at Brunswick intake canal in the refined Cape Fear subdomain.

Figure 2.31 Case 4: Progression of a Brunswick child domain patch for $\tau^0 = 10^{-3}$, $\sigma = 10^{-1}$, $\lambda = 10^{-5}$, and $\theta = 100$, with elements in the patch darkened. Snapshots: (a) initial extent, (b) extent at timestep 1,008,072, (c) largest extent (occurring at timestep 1,203,758), (d) final extent (timestep 1,374,294).

Figure 2.32 Case 4: $l^2$-norms of maximum elevation errors in Brunswick child domains.

Figure 2.33 Case 4: max-norms of maximum elevation errors in Brunswick child domains.

Figure 2.34 Case 4: Average percentage of active elements in Brunswick child domains (with $\lambda$, $\tau$ axes reversed).

Figure 2.35 Case 5: Recording stations at the intake canal.

Figure 2.36 Case 5: Water surface elevations at intake canal recording stations for varying depths and Manning’s $n$ values.

Figure 2.37 Case 6: Local changes on the Cape Fear River.

Figure 2.38 Case 6: Recording stations along the river bank.

Figure 2.39 Case 6: Velocities at recording stations for varying elevation raises and Manning’s $n$ values.

Figure 3.1 Expansion and contraction of locally modified Shinnecock Inlet child domain patch at various timesteps. (Elements included in the patch are highlighted in darker colors.) [3]

Figure 3.2 The simplified class diagram of OpenHDM

Figure 3.3 Main function of ADCIRC++

Figure 3.4 Phase execution criteria for parents and children

Figure 3.5 State diagram of the composition of a parent domain and a child domain with two phases. (Actions with prefixes pr and ch correspond to the ones engaged by the parent and the child, respectively. The indices shown between square brackets show the phase index for the specified domain. The action notify which follows the action run is subjected to hiding operator for brevity.)

Figure 3.6 Flowchart of the timestepping loop of ADCIRC++ domains.

Figure 3.7 Class diagram of ADCIRC++. (All classes are derived from the generic code, except NodalAttr and Wind. Specialized input and output classes are not included.)

Figure 3.8 ADCSolver class inherited from the Solver template. (Attribute accessors and data members are omitted for brevity.)
Figure 3.9 **ADCGrid** class inherited from **Grid** template. (Attribute accessors and data members are omitted for brevity.)

Figure 3.10 The subdomain used as the parent domain and the deepened intake canal for Hurricane Irene simulation

Figure 3.11 Progression of Irene child domain patch for $\tau = 10^{-3}$, $\sigma = 10^{-1}$, $\lambda = 10^{-5}$, $\theta = 100$. Elements included in the patches are highlighted in darker colors. (a) Initial extent. (b) Extent at timestep 1,098,072. (c) Largest extent (occurring at timestep 1,203,758) (d) final extent (reached at timestep 1,374,294)

Figure 4.1 A typical unstructured triangular finite element mesh used in hurricane storm surge modeling, that covers the Western North Atlantic with 620,089 nodes and 1,224,714 elements.

Figure 4.2 Subdomain Modeling Tool (SMT) user interface.

Figure 4.3 Wetting and drying data dependencies [12].

Figure 4.4 A global variable, a SPIN process, and a safety property.

Figure 4.5 Concurrent incrementation.

Figure 4.6 Synchronized incrementation.

Figure 4.7 Synchronized incrementation with a selective structure.

Figure 4.8 A busy-wait implementation using a repetitive statement.

Figure 4.9 A message channel used as a lock.

Figure 4.10 Conditions for an abstract grid to represent a subset of any possible grid.

Figure 4.11 Determining the size of the abstract grid.

Figure 4.12 Abstract grids with minimum and maximum number of interface nodes. Internal and interface nodes are shown in red and gray, respectively.

Figure 4.13 Compound data types: **Node**, **Element**, **Domain**.

Figure 4.14 Macros to retrieve incident nodes of element i.

Figure 4.15 Timestep initialization routine.

Figure 4.16 SPIN model of the first stage of the wetting and drying algorithm.

Figure 4.17 SPIN model of the second stage of the wetting and drying algorithm.

Figure 4.18 SPIN model of the third stage of the wetting and drying algorithm.

Figure 4.19 SPIN model of the fourth stage of the wetting and drying algorithm.

Figure 4.20 Updating permanent wet/dry states of nodes.

Figure 4.21 An inline asserting that elements with three wet nodes must be wet.

Figure 4.22 The initial process of elemental drying verification.

Figure 4.23 The guided simulation of the execution trail that results in the assertion violation.

Figure 4.24 An example configuration for abstract full domain and subdomain grids.

Figure 4.25 The declarations of the boundary conditions channel and inlines.

Figure 4.26 A mechanism for non-deterministic choice synchronization.

Figure 4.27 The modified inline of the third part of the algorithm.

Figure 4.28 The modified nodal loop in **part4(domain)**.

Figure 4.29 The initial process for the verification of wetting and drying in subdomains.
Figure 4.30 The shortest path to the counterexample found in absence of intermediate forcing. (The trail includes additional printf statements for further clarification.) ................................................................. 126

Figure 4.31 Progression of wet/dry states of nodes and elements in absence of intermediate wet/dry forcings for the subdomain. Dry nodes and elements are shown in yellow, while wet nodes and elements are shown in blue. Inactive elements are either dry or has at least one dry node. ................................................................. 127

Figure B.1 Auxiliary inline definitions for the abstract wetting and drying algorithm. . . 145

Figure B.2 Auxiliary macro definitions for the abstract wetting and drying algorithm. . . 146
Chapter 1

Introduction

Hurricane storm surge poses a substantial threat to the built and natural environment in coastal communities. While the devastating impacts of the surge events in recent years, such as the ones induced by Hurricane Katrina (2005) and Hurricane Sandy (2012), remind us of the value of disaster preparedness and risk mitigation efforts, studies on the effects of climate change and sea level rise indicate an increasing risk associated with coastal hazards in the future [73, 89]. In addition to climatic factors, population growth and increasing urbanization in low-lying regions further escalate the vulnerability to inundation [52]. Thus, the prediction and analysis of coastal hazards are highly critical for creating more resilient coastal communities.

An effective method for assessing the impacts of such hazards is the utilization of numerical ocean models. Given a set of inputs including topo-bathymetric data, tidal forces, and meteorological conditions, these models simulate individual and combined effects of coastal events such as tides, hurricane storm surge, and wind waves. Outputs obtained from these models indicate the extent, severity, and timing of simulated inundation, which are used by decision makers for forecasting and hindcasting purposes. The recent use of ADCIRC to provide emergency managers with accurate predictions of waves and circulation during Hurricane Isaac (2012) is an example of the application of ocean models for real-time prediction efforts [35]. In addition to forecasting, numerical ocean models are commonly used for hindcasting and validation studies [6, 44, 57], as well as planning and infrastructure design purposes [29, 50].
Given the spatial and temporal extent of coastal events and complex nature of ocean physics, numerical ocean models require a substantial amount of computational resources. When multiple what-if scenarios are to be evaluated, whether as part of forecasting/hindcasting studies or sensitivity analyses, computational demand further increases, since regardless of the extent of the alterations, each design scenario requires a large-scale simulation to fully capture the physics of coastal processes.

This study presents an adaptive multi-analysis technique for ocean models to reduce the computational effort when multiple local changes are to be evaluated. The technique, called adaptive subdomain modeling, is realized by the concurrent analysis of any number of child domains, each corresponding to a unique design and failure scenario, and an original full-scale parent domain. Based on a prior subdomain modeling approach with static boundary locations, the new approach eliminates the requirement for users to determine the sizes and shapes of modified domains extracted from an original full-scale domain: the spatial extent of each child domain is automatically initialized to cover only the modified regions and is adaptively adjusted during runtime depending on the movement of altered hydrodynamics, i.e., the differences between the solutions of a child domain and its parent domain, originating from the local changes. At every timestep, an error measure indicative of the differences between the solutions of a child domain and its parent domain is calculated around the boundaries to assess the proximity of altered hydrodynamics. If the error indicator is determined to be greater than a user-specified tolerance, then the boundary is moved outward to ensure that the boundary conditions obtained from the concurrent parent domain remain valid. If the error indicator is sufficiently small, then the boundary is moved inward to reduce the computational effort. Thus, the approach ensures that child domain simulations are always accurate and efficient.

The study also includes the development of a modern software architecture for ocean models that can accommodate next-generation approaches like adaptive subdomain modeling. Ocean models are usually based on procedural programming, i.e., they are decomposed into reusable subroutines operating over global and non-reentrant data structures, which prevent effective
incorporation of new features. Our software architecture, however, is based on object oriented programming principles that enhance modularity, maintainability, and extensibility through data abstraction and encapsulation. The architecture, consisting of two levels of source code, facilitates adaptive behavior and utilizes concurrent executions of multiple domain instances. The first level source code is a generic framework consisting of class templates and abstract classes that can accommodate ocean models with different numerical methods and formulations, and the second level is implementation-specific source code that is specialized for the details of the model to be developed. The generic framework also includes a shared-memory parallel implementation for the concurrent execution of multiple domain instances. We demonstrate the effectiveness and flexibility of the architecture by modeling a simple finite-volume shallow water equation solver and by re-implementing ADCIRC, an advanced finite-element ocean circulation model. We incorporate the adaptive subdomain modeling approach in the new implementation, called ADCIRC++, and discuss the advantages of object oriented design in terms of maintainability, extensibility, and computational efficiency.

While object oriented software design facilitates the addition of new computational techniques, the correctness of the algorithmic behavior of an ocean model still needs to be verified for newly acquired features. As an alternative to testing and deductive reasoning, this study proposes the application of model checking tools for the formal verification of algorithmic model behavior. Mainly used for the verification of concurrent hardware and software systems, model checkers analyze the abstract representations of actual systems, which are generated by the users as inputs, and guarantee an effective verification procedure due to their exhaustive state-space exploration strategy. To demonstrate the applicability, the wetting and drying algorithm of ADCIRC is modeled using the SPIN model checker, and the correctness of the algorithm for adaptive subdomain modeling is verified.

The remainder of this dissertation is organized as three separate chapters, each corresponding to a manuscript in preparation that includes one of the main contributions of this dissertation introduced above. In Chapter 2, the adaptive subdomain modeling approach is presented,
and its accuracy and efficiency are evaluated through a series of test cases. In Chapter 3, the new software architecture for concurrent and adaptive ocean models is introduced. Finally, in Chapter 4, the application of model checking for the formal verification of wetting and drying algorithms is described.
Chapter 2

Adaptive Subdomain Modeling: A Multi-Analysis Technique for Ocean Circulation Models

Abstract  Since coastal processes operate over large temporal and geographical scales, ocean models require a substantial amount of computational resources, and even more so when a number of engineering design and failure scenarios are to be considered. This study presents an adaptive multi-analysis technique to improve the efficiency of such computations when multiple scenarios are to be evaluated. The technique, called adaptive subdomain modeling, 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 boundary conditions for the child domains. To contain the altered hydrodynamics originating from the modifications, the spatial extent of each child domain is adaptively adjusted during runtime depending on the response of the model. The technique is incorporated in ADCIRC++, a re-implementation of the ADCIRC ocean circulation model with an updated software architecture designed to facilitate this adaptive behavior and to utilize concurrent executions of multiple domains. The results of various case studies confirm that the method is highly efficient and reliable.
2.1 Introduction

A comprehensive evaluation of the damaging effects of coastal hazards on the built and natural environment requires the assessment of many potential model configurations. While modifications corresponding to design and failure scenarios are often local in nature, ocean processes such as tides and hurricanes operate over significantly larger scales. As a result, despite the limited geographic extent of a region of interest, each local configuration requires a large-scale simulation to accurately capture the physics of the hydrodynamic processes involved [19].

To address this difference in scale, a prior study presented an exact reanalysis technique, called subdomain modeling, that enables the assessment of multiple local changes without requiring a separate full-scale simulation for each one [12]. The technique, implemented in ADCIRC, introduces a new boundary condition type that combines water surface elevation, velocity, and wet/dry status. The workflow begins with the extraction of subdomains from an original full-scale domain. Once subdomain grids are generated, a full-scale simulation is performed to obtain boundary conditions for each subdomain. Local changes corresponding to design and failure scenarios can then be applied to subdomain grids, provided the altered hydrodynamics remain within the boundaries, which are forced with data obtained from the original configuration. The technique substantially reduces the computational effort required to analyze local changes, but requires that users determine a priori the size and shape of each subdomain by anticipating the spatial extent of the effects of those changes. The efficiency of the technique may be reduced when subdomains are oversized, whereas, if undersized, the entire exercise may need to be repeated.

In this study, we present an adaptive subdomain modeling (ASM) technique where the sizes and shapes of computationally active regions, called patches,\(^1\) of locally modified grids are automatically determined and adaptively adjusted during runtime. The technique is real-

\(^1\)The term patch has a variety of definitions depending on context. For instance, in GeoClaw, a finite-volume hydrodynamic model with adaptive refinement capability [17], the term corresponds to overlapping layers of a computational grid with different refinement levels. Here we use the term to refer to computationally active regions of a grid.
ized by concurrently executing the simulations of multiple child domains, with each instance corresponding to a local scenario, and a full-scale parent domain providing the boundary conditions for the child domains. Initially encompassing only the modified regions, the patches of child domains are dynamically adjusted during runtime depending on the response of the model. An error indicator—a measure of the difference between the solutions of the parent and child domains—is calculated near the boundaries of patches to assess the proximity of altered hydrodynamics to the boundaries. In case the error indicator is determined to be larger than a user-specified tolerance, the patch boundary at that location is moved outward to ensure that the changing hydrodynamics approaching the boundary do not reach it before the next timestep. If the error indicator is determined to be sufficiently small, on the other hand, the patch is contracted to increase computational efficiency. An example of a child domain as a dynamically adjusting patch is shown in Fig. 3.1.

To accommodate the ASM approach, we make use of a re-implementation of ADCIRC with an updated software architecture that more readily supports adaptivity. In its original form, ADCIRC is based on procedural decomposition, with code that is structured by dividing control flow into subroutines, and where the primary data structures are global and non-reentrant. Our new architecture, on the other hand, is based on data abstraction, where the internal representation of a data type is distinct from its external view. This style of programming enhances modularity, maintainability, and extensibility by ensuring that the changes made within a data structure do not propagate to the rest of the program [13]. The new implementation, called ADCIRC++, facilitates adaptive grid behavior and utilizes concurrent executions of multiple domains by means of dynamic containers and object-oriented design principles [4].

The remainder of the paper is organized as follows. In Section 2, we briefly describe ADCIRC and our original subdomain modeling approach, hereafter referred to as conventional subdomain modeling (CSM). In Section 3.2.4, the integral components of our new ASM approach are described: the error indicator, adaptivity algorithm, and application of boundary conditions. In Section 2.4, implementation details of ASM are presented, along with differences between ASM
and CSM workflows and a hybrid approach that combines the two. Section 2.5 includes parametric studies with test cases that serve as a guide for determining the ASM control parameter settings subsequently used, and a sensitivity analysis that demonstrates the applicability and computational efficiency of the method. Finally, conclusions and future work are presented.

2.2 Background

2.2.1 ADCIRC

ADCIRC is a continuous Galerkin finite element ocean circulation model, widely used by the US Army Corps of Engineers (USACE), Federal Emergency Management Agency (FEMA), and various other agencies and institutions to simulate tides and hurricane storm surge [90].
Combined with the flexibility of unstructured triangular meshes, ADCIRC’s formulation of the
governing equations and optimized numerical algorithms constitute an efficient and versatile
modeling system [74].

As for the computational process, at every timestep ADCIRC solves the generalized wave
continuity equation (GWCE) to obtain water surface elevations, then executes a wetting and
drying algorithm to determine the geographic extent of hydrodynamic activity, and finally solves
the momentum equations to obtain velocities in the $x$ and $y$ directions.

ADCIRC simulations can be performed as three dimensional or two dimensional depth
integrated (2DDI) analyses. The linear system of the GWCE can be configured so that it is
based on either consistent or lumped mass matrices, and time discretization may be performed
either implicitly or explicitly. The consistent GWCE system is solved using an iterative Jacobi
conjugate gradient method. For a 2DDI model with a consistent matrix solver and implicit
timestepping scheme—as used in this study—both the GWCE and the momentum equations are
discretized in space using the Galerkin finite element method [90], and the GWCE is discretized
in time using a variably weighted three-time-level implicit scheme for the linear terms, while
the momentum equations are discretized in time using a two-time-level implicit Crank-Nicolson
approximation [74].

2.2.2 Conventional subdomain modeling and applications

A basis for our adaptive technique is CSM, a static precursor that similarly enables the as-
seSSment of local alterations with less computational effort than would be required by repeated
simulations on a full-scale grid [12]. Local changes can be applied to subdomain grids once they
are extracted from an original full-scale grid to simulate design and failure scenarios, provided
the subdomains are large enough to fully contain the altered hydrodynamics. The locations
of the static boundaries of subdomain grids are predetermined by the user and enforced using
boundary conditions that are defined by elevations, velocities, and wet/dry states obtained from
the outputs of the original full-scale simulation.
The CSM workflow is as follows:

1. Construct a subdomain

   1.1. Locate one or more regions of interest within the original full domain

   1.2. Perform a simulation on the full domain to generate boundary conditions for each subdomain

   1.3. Preprocess boundary condition files

   1.4. Perform simulations on subdomains as a verification step

2. Generate engineering scenarios

   2.1. Alter subdomains to realize engineering design and failure scenarios of interest

   2.2. Perform simulations on altered subdomains

   2.3. Check results of altered subdomains as a second verification step

The two verification steps in the workflow are performed to confirm that results from unaltered subdomains match their full domain counterparts, and that the new hydrodynamics induced by altered subdomains do not propagate to subdomain boundaries.

The pre- and post-processing steps of CSM are facilitated by a graphical user interface called SMT, the Subdomain Modeling Tool [42]. Multiple subdomains can be visually extracted using a variety of selection tools, as shown in Fig. 2.2. Once subdomains are defined by the user, the tool automatically generates the required input files for both the subdomains and the full domain.

CSM is incorporated in the official ADCIRC release, beginning with v51.42, and is now in active use by the modeling community. In one application of CSM, Butler et al. [26] determine spatially varying Manning’s n values probabilistically by formulating and solving a stochastic inverse problem. They employ a measure-theoretic framework to address the issue of uncertainties in the inverse problem due to the mapping from parametric data (Manning’s n) to observational data (maximum water surface elevation), and due to errors in measurements. Once
Manning’s n fields are determined probabilistically, the results can then be used for predictive simulations, which may easily become computationally prohibitive. They point out, however, that the use of subdomain modeling can reduce the computational time and allow focusing on specific regions of interest that are prone to hurricane storm surge. Subsequently, Graham, one of the co-authors, reduces the cost of a series of forward models using the subdomain modeling approach for his Hurricane Gustav Case Study [47], where he extracts a subdomain grid with 15,001 elements from a full-scale grid consisting of 2,720,591 elements. He notes that the runtime required for the full-scale grid is about 3,300 CPU-hours, whereas for the subdomains it is only 11 CPU-hours.

In another application of CSM, Haddad et al. [51] investigate the factors affecting the behavior of storm surge in wetlands by combining field work and numerical modeling. To assess the effects of landscape conditions and surface roughness on water levels, velocities, and wind fields, for instance, they carry out ADCIRC+SWAN simulations with varying Manning’s n values, directional surface roughness length coefficients, and dense tree canopies. Since such
sensitivity studies require substantial computational resources, they remark on the anticipated value of subdomain modeling in reducing the cost of repeated simulations with adjustments to the grid and the vegetation parameters in the regions of interest.

2.3 Adaptive subdomain modeling

ASM is an improved and complementary technique for ocean models that allows the simulation of locally modified child domains to be performed concurrently. Such modifications, for instance, might include changes to bathymetry, bottom friction, and elemental slope limiters that constitute an alternative modeling scenario. By adaptively moving boundaries that are forced with data from the parent, the technique avoids performing computations that are external to a child domain and therefore redundant.

The ASM approach consists of three essential components. First, an error indicator determines the progression of altered hydrodynamics. Second, an adaptivity algorithm for the expansion and contraction of patches manages the insertion and removal of nodes and elements. Finally, boundary conditions are prescribed for accurate computations within child domain patches. In our implementation, the original ADCIRC timestepping loop is modified so that the adaptivity algorithm is executed at the beginning of each timestep to determine and apply any necessary adjustments to patch boundaries. Then, boundary conditions are enforced at specified control points as is done in the CSM approach. A flowchart of the modified timestepping loop for ASM is shown in Fig. 2.3.

2.3.1 Error indicator

The decision of whether to move a patch boundary is based on an error measure indicative of the altered hydrodynamics, i.e., the differences between the water surface elevations and velocities of the child domains and the parent domain. In other applications, such as adaptive mesh refinement (AMR), error indicators determine how to adjust computational grids, determining where to refine or coarsen them to reduce the numerical error or to increase the computational
Figure 2.3: Flowchart of the modified timestepping loop for adaptive subdomain modeling (ASM steps shaded, steps common to ASM and CSM patterned, and original steps left unshaded).
efficiency. Such indicators may be based on solutions [14, 43, 66], derivatives of solutions [72, 91], mass residuals [91, 36, 78], or truncation errors [16, 18]. Thus, their purpose is to guide decisions about mesh refinement and corresponding numerical schemes in an effort to improve overall convergence. In ASM, by way of contrast, the objective is to detect the altered hydrodynamics originating from local changes, and thereby ensure that each locally modified child domain behaves as though it were part of its own full-scale domain in a full-scale simulation. As a result, an error indicator based on differences between the solutions of child domains and parent domains is chosen:

$$\rho = \max(\rho_\eta, \rho_u, \rho_v)$$

where

$$\rho_\eta = \left( \frac{|\eta_{\text{child}} - \eta_{\text{parent}}|}{\sqrt{0.5(|\eta_{\text{child}}| + |\eta_{\text{parent}}|)}} \right)^2$$

$$\rho_u = \left( \frac{|u_{\text{child}} - u_{\text{parent}}|}{\sqrt{0.5(|u_{\text{child}}| + |u_{\text{parent}}|)}} \right)^2 \Delta t$$

$$\rho_v = \left( \frac{|v_{\text{child}} - v_{\text{parent}}|}{\sqrt{0.5(|v_{\text{child}}| + |v_{\text{parent}}|)}} \right)^2 \Delta t$$

\(\eta\): water surface elevation

\(u\): \(x\) velocity

\(v\): \(y\) velocity

\(\Delta t\): step size in seconds

Among several forms we have experimentally evaluated, the error indicator shown proves to be both stable and efficient for a wide range of model configurations. For instance, compared with absolute difference as an indicator, this form provides more sensitivity for smaller magnitudes of errors relative to larger ones. As used within our analysis procedure, the \(\rho\) indicator in Equation (2.1) is calculated at nodes adjacent to the boundaries of child domain patches. The tolerance of a patch boundary node—a control parameter initially set by the user—is then com-
pared with the error indicators of the adjacent nodes to determine whether the patch boundary should be moved at that location.

### 2.3.2 Adaptivity algorithm

In the ASM approach, child domain patches are first initialized to include only the nodes whose properties have been modified as part of an alternative modeling scenario, along with a three-layer buffer of surrounding nodes and elements, as shown in Fig. 2.4. Each layer has a rationale: the first is adjacent to and directly affected by changes to modified nodes, the second assesses the potential for altered hydrodynamics, and the third enforces boundary conditions obtained from the parent domain. Once the initial patch of nodes and elements has been determined and *activated* for each child domain, the corresponding systems of equations are constructed, and simulation can begin. Then, during runtime, child domain patches are adaptively adjusted to ensure that they are just large enough to cover the altered hydrodynamics. Control parameters that determine the shapes and sizes of patches are as follows:

**tolerance** (*τ*): a parameter that varies with timestep and against which error indicators are compared to determine whether a patch expands; the comparison is of the form \( \rho > \tau \). An initial tolerance of \( \tau^0 \) is set by the user, and subsequent changes are made as necessary by the adaptivity algorithm.

**minimum activation interval** (*θ*): the minimum number of timesteps throughout which a newly activated node must stay active; nodes within a patch are referred to as *active*.

**decay constant** (*λ*): a parameter controlling the exponential decay of tolerance \( \tau \) based on a reduction of the form \( e^{-\lambda} \). Such reductions are applied after an increase in tolerance to return it over time to its initial setting, \( \tau^0 \).

**contraction factor** (*σ*): a constant set by the user that, along with the initial tolerance \( \tau^0 \), is compared with error indicators to determine whether a patch contracts; the comparison is of the form \( \rho < \sigma \tau^0 \) and assumes that \( \sigma \) is less than one.
As an illustration of the relationship between control parameters, Fig. 2.5 shows a time history of the maximum error indicator at a timestep and its effect on the maximum tolerance for a hypothetical child domain. Each time the tolerance is exceeded by the error indicator near a patch boundary, the boundary is moved outward one layer, and the tolerance of the newly expanded boundary node \( (e) \) is set to the sum of the initial tolerance and the error indicator value of the marked node \( (m) \) causing the expansion; in other words, \( \tau_{e}^{t+1} = \tau^{0} + \rho_{m}^{t} \). Increasing the tolerance affords local errors some time to decrease without causing the boundary to be moved outward repeatedly at consecutive timesteps. Locally increased tolerances return to their initial setting over time based on the user-specified exponential decay constant \( (\lambda) \).

**Boundary expansion and numerical stability**

Before elaborating on the stages of the adaptivity algorithm, we consider the relationship between the boundary expansion procedure and the stability of the numerical scheme, which taken together must ensure that altered hydrodynamics are contained within the patches of child domains throughout the simulation.

The process of expanding patches relies on the assumption that the underlying numerical method employed, in this case by ADCIRC, satisfies the Courant-Friedrichs-Lewy (CFL) condi-
Figure 2.5: Relationship between tolerance and error indicator during a simulation. The patch expands at timesteps $t_1$, $t_3$, and $t_4$, and contracts at timestep $t_2$ since $\rho_{max}$ falls beneath $\sigma r^0$.

tion, which is necessary for the convergence of hyperbolic PDEs. The CFL condition states that a method can only be convergent if the numerical domain of dependence encompasses the analytical domain of dependence of the PDE [69]. Since hyperbolic PDEs have a finite information propagation speed, their domain of dependence is finite, i.e., the solution at a node depends only on a finite domain [54]. The CFL condition is tested by comparing the Courant number, a ratio of $\Delta t$ to $\Delta x$, against an upper bound. For ADCIRC and its semi-implicit time marching algorithm, the Courant number should be at most 0.5 for open ocean flows, and much less for other situations like near-shore flows with wetting and drying [40]. It is defined as:

$$C_r = \frac{\sqrt{gh} \Delta t}{\Delta x}$$  \hspace{1cm} (2.2)

where $\sqrt{gh}$ is the linear wave celerity, $\Delta t$ is the step size, and $\Delta x$ is the distance between two nodes. Note that the given Courant number does not account for velocity but only for celerity since, for ADCIRC simulations, celerity is almost always expected to be greater than velocity by at least an order of magnitude, and hence is more limiting.
In summary, the CFL condition limits the maximum stable step size for a computational grid so that the solution at any point does not propagate beyond the domain of dependence, i.e., one layer of elements, within a timestep. This restriction also ensures that expanding a child domain by a single layer of elements is sufficient to contain the altered hydrodynamics: such changes are guaranteed to propagate no further than a layer at a time, and therefore cannot reach the boundary. Once differences are detected at a node, the patch is expanded so that two layers separate any such nodes from the patch boundary.

As an alternative to the semi-implicit time-marching algorithm, one might consider using an implicit scheme so that the Courant stability constraint can be relaxed [41]. In ADCIRC, however, the wetting and drying algorithm imposes an additional restriction on step size, since wetting fronts can propagate only one layer per timestep [33]. As a result, regardless of the time-marching algorithm employed, the step size must be small enough so that the solution is limited to advancing a single layer of elements at a time, further justifying the ASM expansion policy in practice.

**Stages of the adaptivity algorithm**

The adaptivity algorithm consists of four main stages that are executed in turn at the beginning of each timestep. In the first, the algorithm calculates the error indicator \( \rho \) near patch boundaries, marks areas where a tolerance is exceeded for expansion, and areas where the indicators are sufficiently small for contraction. In the remaining three stages it performs the expansions and contractions, and finally carries out a post-processing step to update the affected properties and data structures. Implementation details of each stage are given below.

**Stage 1: Assessment of altered hydrodynamics** The algorithm begins by evaluating criteria that determine locations for expansion and contraction on patch boundaries. For expansion, error indicators at nodes adjacent to patch boundaries provide a way to gauge whether hydrodynamic changes are impinging, as signaled when indicator \( \rho \) exceeds tolerance \( \tau \). Such
nodes are marked for expansion in a subsequent stage. As patches grow, they may eventually coincide with certain types of boundaries defined in the parent domain, such as mainland or island boundaries. However, if they reach boundary types defined by flux or water surface elevation, such as an open ocean boundary, execution of the child domain is aborted since it will have failed to satisfy the specified tolerance. The criteria for expansion are summarized in Fig. 2.6 and illustrated in Fig. 2.7.

**Expansion criteria:** mark a patch boundary node $i$ for expansion if

- $\exists n \in \text{ineitab}(i): \rho_n > \tau_i$ and
- $i$ not adjacent to a grid boundary (except island or mainland)

where $\text{ineitab}(i)$ is the set of neighboring internal nodes of $i$.

Figure 2.6: Summary of criteria for expanding a patch.

Figure 2.7: Illustration of criteria for expanding a patch.
For contraction, several criteria come into play. The first is that all internal neighbors of a patch boundary node must have error indicator values $\rho$ less than $\sigma \tau^0$, the product of the contraction factor and the initial tolerance. Then, the same test must be satisfied by the neighbors’ neighbors of the boundary node, since nodes adjacent to the boundary node are about to become boundaries themselves (once the boundary node under consideration is deactivated). Continuing with other criteria, the minimum activation interval $\theta$ must be satisfied to prevent flickering of the nodes and elements, as is done in AMR implementations [66]. Finally, the boundary node under consideration should not be adjacent to a node marked for expansion, and it should not be one of the initially active nodes in the original patch. If these hold, such nodes are marked for contraction in a subsequent stage and added to a deactivation list. The criteria for contraction are summarized in Fig. 2.8 and illustrated in Fig. 2.9.

**Contraction criteria:** mark a patch boundary node $i$ for contraction if

- $\forall n \in \text{initab}(i) : \rho_n < \sigma \tau^0$ and $\forall m \in \text{initab}(n) : \rho_m < \sigma \tau^0$ and
- $i$ activated at least $\theta$ timesteps ago and
- $i$ not adjacent to an expansion node and
- $i$ not included in the initial patch

where $\text{initab}(i)$ is the set of neighboring internal nodes of $i$.

Figure 2.8: Summary of criteria for contracting a patch.

**Stage 2: Expansion** After marking nodes for expansion, the algorithm is ready to make changes to the patch by moving boundaries outward at those locations. The expansion stage of the algorithm is presented in Fig. 2.10, which converts the marked boundary nodes to internal nodes, determines which external nodes comprise the expansion layer, activates them and the elements incident on them, updates the topology (connectivity) of the grid, and sets the
tolerances of new patch boundary nodes. Fig. 2.11 illustrates the approach.

With respect to memory management, optimizations are performed to minimize repetitive allocation and deallocation of nodes and elements. When a node external to a patch is activated for the first time, for instance, space is allocated for it as part of the child domain. The node is also marked as active, but at a later time it might be deactivated, at which point it is removed from the child domain but its underlying space allocation remains. If it is subsequently activated, then, no reallocation of space is required.

After the designated nodes and elements are activated, some bookkeeping and clean-up steps must be performed. The connectivity of all nodes and elements affected by the activations is updated. Patch boundary nodes not marked for expansion that are surrounded by newly activated nodes and elements are converted to internal nodes to prevent them from being treated as boundary conditions and checked for expansion or contraction. Finally, the tolerances of the new boundary nodes are updated.

**Stage 3: Contraction**  Presented in Fig. 2.12, the contraction stage of the algorithm includes the following major steps: clean up the list of nodes marked for deactivation in the first stage,
Stage 2: Expansion Algorithm

- Let `expansionNodes` be the set of patch boundary nodes where the boundary is to be expanded. The set is assembled in Stage 1
- Let `nodesToAllocate` and `nodesToActivate` be empty sets of nodes
- Let `neighbors(i)` be the set of nodes adjacent to node `i`
- Let `elements(i)` be the set of elements incident on node `i`
- Let `nodes(i)` be the list of nodes of element `i`

1: for `i` in `expansionNodes` do
   Convert `i` to internal node  \(\triangledown\) to remove it from the set of patch boundary nodes
   for `j` in `neighbors(i)` do
     if `j` is not instantiated before then
       `nodesToAllocate.insert(j)`  \(\triangledown\) copy the external neighbor from parent grid
     else if `j` is inactive then
       `nodesToActivate.insert(j)`  \(\triangledown\) (i.e., copied at a previous timestep)

2: for `i` in `nodesToAllocate` do
   Copy `i` from parent and insert to patch
   if `i` is mainland boundary or island boundary then
     Update the relevant vectors and parameters

3: for `i` in `nodesToActivate` do
   Insert `i` to patch  \(\triangledown\) and so activate `i`
   Copy time-varying properties of node `i` from parent

4: for `i` in (`nodesToActivate+nodesToAllocate`) do
   for `j` in `elements(i)` do
     \(\triangledown\) loop through elements connected to `i`
     if (`j` is not instantiated before) and (all three nodes of `j` are instantiated) then
       Copy `j` from parent
     if (`j` is inactive) and (all three nodes of `j` are active) then
       Insert `j` to patch  \(\triangledown\) and so activate `j`
       for `k` in `nodes(j)` do
         \(\triangledown\) loop through nodes of element `j`
         Update `elements(k)`

5: for `i` in (`nodesToActivate+nodesToAllocate`) do
   Update `neighbors(i)`
   for `j` in `neighbors(i)` do
     Update `neighbors(j)`

6: for `i` in (`nodesToActivate+nodesToAllocate`) do
   if `i` is connected to at least one inactive node then
     Convert `i` to boundary node
   else
     Convert `i` to internal node  \(\triangledown\) i.e., surrounded by active nodes

Figure 2.10: Expansion stage of the adaptivity algorithm.
Figure 2.11: Main steps of the expansion of a child domain patch, where the patch boundary node $r$ is marked for expansion at the first step of the algorithm.

process that list by actually deactivating nodes in the child domain, and update the connectivity of nodes and elements. Fig. 2.13 illustrates the approach.

Before any nodes are deactivated, the algorithm checks for inconsistencies. As a result of the marking in stage 1, some of the patch boundary nodes may become disconnected from internal nodes and remain connected only to patch boundary nodes; such nodes are now added to the deactivation list. Conversely, some nodes are removed from the deactivation list, namely, those that are surrounded by active nodes as a result of expansions, and those that are adjacent to an expansion node or a recently activated node.

At this point, nodes are deactivated by removing them from the patch. Internal nodes that are connected to deactivated nodes are then converted to patch boundary nodes. Additionally, elements incident on deactivated nodes are deactivated by removing them from the patch, and nodal connectivity is updated. Once this process is complete, nodes that are only incident on inactive elements are deactivated. Finally, the connectivity of all affected nodes and elements is updated.
Stage 3: Contraction Algorithm

- Let `contractionNodes` be the set of patch boundary nodes where the boundary is to be contracted. This set is assembled in Stage 1
- Let `patch.boundaryNodes` be the set of boundary nodes of `patch`
- Let `deactivatedElements` be an empty set of elements
- Let `neighbors(i)` be the set of nodes adjacent to node `i`
- Let `elements(i)` be the set of elements incident on node `i`
- Let `nodes(i)` be the list of nodes of element `i`

1: for `i` in `patch.boundaryNodes` do
   if `(i` is connected only to patch boundary nodes) and `(i` is not initially active) then
       `contractionNodes.insert(i)`

2: for `i` in `contractionNodes` do
   if `(i` is not at the boundary anymore) or `(i` is next to an expansion node) or `(i` is next to a recently activated node) then
       `contractionNodes.remove(i)`

3: for `i` in `contractionNodes` do
   Deactivate `i`
   Convert `i` to internal node  ▶ to remove it from the set of patch boundary nodes
   for `j` in `neighbors(i)` do
     Update `neighbors(j)`
     if `(j` is internal node) then
       Convert `j` to boundary node
   for `k` in `elements(i)` do
     Deactivate `k`
     `deactivatedElements.insert(k)`
   for `l` in `nodes(k)` do  ▶ loop through nodes of element `k`
     Update `elements(l)`

4: for `i` in `deactivatedElements` do
   if any node `j` of `i` is not connected to any active element then
     Deactivate `j`
     Convert `j` to internal node  ▶ to remove it from the set of patch boundary nodes
   for `k` in `neighbors(j)` do  ▶ to remove it from the set of patch boundary nodes
     Update `neighbors(k)`
     if `(k` is internal node) then
       Convert `k` to boundary node
   if any two nodes of `i` are disconnected then
     Update `neighbors` of both nodes

Figure 2.12: Contraction stage of the adaptivity algorithm.
Figure 2.13: Main steps of the contraction of a child domain patch, where the patch boundary node \( r \) is marked for contraction at the first step of the algorithm.

**Stage 4: Post-processing**  After the expansion and contraction processes are complete, properties and containers of the child domains are updated in this final stage of the algorithm. As part of that process, the system of equations associated with any expanded or contracted patch is reset and resized. Then, auxiliary containers holding nodal data are updated. Finally, patch boundary nodes with tolerances greater than \( \tau^0 \) are subjected to an exponential decay so that their individual tolerances converge toward the initial tolerance over time. For a patch boundary node \( j \) at timestep \( t \), the tolerance is updated as follows:

\[
\tau_j^t = (\tau_j^{t-1} - \tau^0)e^{-\lambda} + \tau^0
\]

(2.3)

where \( \lambda \) is the decay constant, \( \tau_j^{t-1} \) is the tolerance of boundary node \( j \) at the previous timestep, and \( \tau^0 \) is the initial tolerance.
2.3.3 Boundary conditions

To perform simulations concurrently, an interface is needed between a parent domain and its children. For its basis, we adapt the boundary condition type used in CSM, which incorporates water surface elevation, wet/dry status, and velocity, to realize a one-way hand off from parent to child [12]. The conventional approach to subdomain modeling obtains these quantities after completion of a full-scale run, and then applies them to static boundaries of a subdomain. In ASM, of course, boundaries are in motion, but only at the start of a timestep, giving us a static snapshot afterward in which boundary conditions may be applied. To do so, we (a) specify nodal elevations in the implicit GWCE formulation, (b) force wet/dry status on boundary nodes in the wetting and drying routine, and (c) assign boundary velocities outright in the momentum equation solver.

Of the three conditions—water surface elevation, wet/dry status, and velocity—the enforcement of nodal wetting is somewhat less straightforward because, during each timestep, ADCIRC’s wetting and drying algorithm [37] performs several updates to a node before its final wet/dry state is set, and these intermittent changes are spatially dependent on the intermediate states of other, neighboring nodes. In a prior study [12], we present an analysis of data dependencies and interactions between the wetting and drying algorithm and the hand off required by subdomain modeling and other mesh partitioning schemes. Included is a proof showing that, for correctness, the multiple intermediate wet/dry states of a subdomain boundary node can be set with a single value: the node’s final wet/dry state at a given timestep from a full run.\(^2\) The implication is that the only data transfer required from one domain to another is that of the final wet/dry states, simplifying communication between domains. Applying this result to ASM, we again note that patch boundary nodes are spatially adjusted at the beginning of a timestep and otherwise remain fixed throughout its execution. Thus, apart from extraction and processing procedures, boundary conditions in ASM can be enforced in the same manner as they are in CSM.

\(^2\)Verification of the same results using software model checking techniques can be found elsewhere [11, 5].
2.4 Workflow and hybrid approach

Using ASM begins with a modeling step: identifying geographic locations of interest and determining the alternatives to be simulated in concert with an ordinary ADCIRC model. Then, a single input file for each child domain is created: a difference file (\texttt{.dif}) containing a list of modified nodes along with new values of their associated properties, e.g., bathymetry, bottom friction, and elemental slope limiters. In contrast with CSM, subdomain boundaries are not defined, and the abbreviated versions of input files ordinarily used by subdomains are not required. Instead, the locations, sizes, and shapes of initial child domain patches are determined automatically, and model parameters are copied as needed from the parent domain.

With respect to other input files, we rely on standard ADCIRC file formats and add some of our own. To organize them, we define the notion of a \textit{project} to be an ordinary ADCIRC model plus some number of child domains: a project file (\texttt{.prj}) contains a list of the included domains, i.e., the parent and all of its children. Each domain included in a project file has an associated configuration file (\texttt{.cfg}) that points to the locations of standard ADCIRC files used by the domain and a difference file. An optional input file for child domains is the ASM file (\texttt{.asm}), where control parameters for adaptive subdomain modeling are set; if missing, default values are assumed.

In addition to being straightforward, the ASM workflow eliminates two verification steps that are required by CSM: confirming the stability of unaltered subdomain grids, and ensuring that altered hydrodynamics do not reach subdomain boundaries. Fig. 2.14 summarizes the complete workflow of a typical ADCIRC++ run with ASM.

As an optional step in the workflow, users can experiment with ASM control parameters. Although ADCIRC++ provides a set of default values, the efficiency and accuracy of the technique can be optimized by varying them and examining their effects. Doing so requires only a single concurrent execution of a parent domain and some number of child domains with different control parameter settings, so the additional cost is marginal.
0. Begin with an ordinary ADCIRC model

1. Generate ADCIRC++ input files:
   • a difference file (.dif) with modified nodes for each child domain
   • a project file (.prj) that lists parent and child domains
   • a configuration file (.cfg) for each domain that points to standard ADCIRC files, a difference file, and an optional ASM file (.asm)

2. Run ADCIRC++

Figure 2.14: The complete workflow of a typical ADCIRC++ run with ASM.

While ASM offers some important advantages, an apparent weakness is the inability of users to alter one or more child domains after reviewing the results produced by another, unless they resort to running another full-scale simulation. However, ASM and CSM are complementary techniques that can be used in combination in cases such as the above, which call for a sequential analysis of subdomains. Using CSM, a single, full-scale simulation can produce one or more conventional, static subdomains, any of which can be used as the parent domain in an ASM simulation with any number of child domains. We refer to this combination as hybrid subdomain modeling (HSM), and present examples of its use below.

2.5 Test cases

In this section, we present the results of two sets of parameter studies on realistic application domains. The first set focuses on ASM control parameters, and for that we perform simulations at four different sites, include a single alternative scenario for each, and vary control parameters to analyze their effects on the accuracy and efficiency of the method. We consider both astronomical tide and meteorological forcing examples. In the second set, we look at applications of ASM to storm surge problems at two sites, varying bathymetric depths and bottom friction values parametrically while using control parameter settings informed by the first study.
In all cases, errors are determined by comparing ASM results with a separate, independent run of a parent domain that directly incorporates the local change previously simulated by its children.

2.5.1 ASM control parameters

For evaluating the effects of ASM control parameters on accuracy and efficiency, we look at the following four cases:

1. Shinnecock Inlet on the south shore of Long Island, NY
   A tidal model with a coarse grid, limited area, and a bathymetric change in the inlet

2. Hatteras Inlet located along the Outer Banks, NC
   A tidal model with a finer grid, spanning North Carolina, and a bathymetric change in the inlet

3. Walden Creek at Southport, NC
   Hurricane Fran (1996) simulated on a subdomain around Cape Fear, with an added protective structure

4. Brunswick intake canal at Southport, NC
   Hurricane Irene (2011) simulated on a more refined subdomain around Cape Fear, with depth added all along the Brunswick nuclear power plant’s intake canal

Each case is simulated with a number of concurrent child domains, where each child has the same local change but different values of ASM control parameters. The range of control parameter settings presented reflects a limit for each domain: tolerances smaller than the smallest tolerance and decay constants larger than the largest decay constant result in child domain patches reaching an open ocean boundary, thereby causing the technique to fail.
Case 1: Shinnecock Inlet with tidal forcing

As an introductory example, a tidal model developed by the USACE Coastal Hydraulics Laboratory [80, 81, 95] and available on the ADCIRC website [2] is used. Centered on Shinnecock Inlet, New York, the model is realistic, though coarse in time and space, with a grid of 5,780 elements and 3,070 nodes covering a small area. The total duration of the simulation is 5 days, and the step size is 6 s. Tidal constituents M2, N2, S2, K1, and O1 are applied as tidal potential forcings and tidal boundary forcings. To simulate a small, local change, the bathymetric depths of three nodes near the inlet are reduced by 1 m, as shown in Fig. 2.15.

As the simulation unfolds, child domain patches with their local changes expand to contain the altered hydrodynamics. For ASM control parameter settings of $\tau^0 = 10^{-3}$, $\sigma = 10^{-2}$, $\lambda = 10^{-4}$, and $\theta = 10$, for instance, the associated patch reaches its maximum size of 910 elements (about 16% of the entire grid) at timestep 2,949 (about 4% of simulated time), as shown in Fig. 2.16. The size of the patch remains mostly the same throughout the simulation, since expansions and contractions come into equilibrium once it covers the maximum region of altered hydrodynamics.

For the parametric study, 120 ($= 5 \times 3 \times 4 \times 2$) child domains are concurrently simulated using all combinations of the following control parameter settings: $\tau^0 = \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$, $\sigma = \{10^{-1}, 10^{-2}, 10^{-3}\}$, $\lambda = \{10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}\}$, and $\theta = \{1, 10\}$. The results of modified

![Figure 2.15: Case 1: Local change near Shinnecock Inlet.](image-url)
child domains are compared with those of the parent domain from a separate run after the same local change has been made to it. Figs. 2.17 and 2.18 show the $l^2$-norms and max-norms of errors in maximum elevations for each setting. Fig. 2.19 shows the average percentage of active elements for each child domain.

As seen in the graphs, the initial tolerance setting has the greatest influence on the accuracy of the approach for this simple model. The largest improvements in accuracy for both the $l^2$-norms and max-norms are observed when reducing the initial tolerance from $10^{-2}$ to $10^{-3}$. Effects of adjustments to the remaining parameters are less significant.

**Case 2: Hatteras Inlet with tidal forcing**

A tidal circulation problem presented by Luettich et al. [75] evaluates larval transport characteristics on the North Carolina coast. Available from the ADCIRC website [2], the *Beaufort Inlet* grid file used in that study consists of 32,218 nodes and 58,641 elements, and spans the
Figure 2.17: Case 1: $l^2$-norms of maximum elevation errors in Shinnecock child domains.
Figure 2.18: Case 1: max-norms of maximum elevation errors in Shinnecock child domains.
Figure 2.19: Case 1: Average percentage of active elements in Shinnecock child domains (with λ, τ axes reversed).
coast of North Carolina. We use the grid to perform a 40-day run with a step size of 5 s that includes 5 tidal constituents: K1, O1, M2, N2, and S2. To simulate a local change, we increase the depths of 20 nodes in Hatteras Inlet by an average of 1.425 m, as shown in Fig. 2.20.

![Figure 2.20: Case 2: Local change at Hatteras Inlet.](image)

Child domain patches expand primarily during the first five days of the simulation, which coincides with the ramp function ADCIRC uses to avoid exciting resonant modes due to a cold start [94]. Fig. 2.21 shows the extent of the Hatteras child domain patches at various timesteps for ASM control parameter settings of $\tau^0 = 10^{-3}$, $\sigma = 10^{-2}$, $\lambda = 5 \times 10^{-4}$, and $\theta = 100$. The change in patch sizes is minimal once it achieves its maximum extent.

For the parametric study, 80 child domains are concurrently simulated using all combinations of the following control parameter settings: $\tau^0 = \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$, $\sigma = \{10^{-1}, 10^{-2}\}$, $\lambda = \{5 \times 10^{-3}, 5 \times 10^{-4}, 5 \times 10^{-5}, 5 \times 10^{-6}\}$, and $\theta = \{10, 100\}$. As before, the results of modified child domains are compared with those of the parent domain from a separate run after the same local change has been made to it. Figs. 2.22 and 2.23 show the $l^2$-norms and max-norms of errors in maximum elevations for each setting. Fig. 2.24 shows the average percentage of active elements for each child domain.

Similar to the Shinnecock case, the accuracy of the approach is mostly influenced by the initial tolerance. Here, however, the decay constant also has some effect, especially for an initial
Figure 2.21: Case 2: Progression of a Hatteras child domain patch for $r^0 = 10^{-3}$, $\sigma = 10^{-2}$, $\lambda = 5 \times 10^{-4}$, and $\theta = 100$, with elements in the patch darkened. Snapshots: (a) initial extent, (b) expansion at timestep 1905, (c) largest patch, and (d) final extent (timestep 691 196).

Figure 2.22: Case 2: $l^2$-norms of maximum elevation errors in Hatteras child domains.
tolerance of $10^{-3}$ or $10^{-4}$. The remaining parameters are less influential.

**Case 3: Walden Creek and Hurricane Fran (1996)**

As an example of meteorological forcing that also happens to use conventional subdomains, a large-scale storm surge model from a prior study [12] is simulated using HSM, a hybrid approach to subdomain modeling that combines ASM and CSM. The full-scale grid from that study consists of 620,089 nodes and 1,224,714 elements encompassing the western North Atlantic Ocean, the Caribbean Ocean, and the Gulf of Mexico. Along the coastlines are external land boundaries having no normal flow and free tangential slip, and along the eastern edge of the domain is a steady open ocean boundary condition.

The specified nodal attributes include surface directional effective roughness length, Manning’s $n$ at the sea floor, surface canopy coefficient, and primitive weighting in the continuity equation. For Hurricane Fran, a 0.5-s step size is used to perform a 3.9-day simulation of the event as a 2DDI analysis.

To employ HSM, we first perform a run on the full domain to generate boundary conditions for a circular subdomain consisting of 28,643 nodes and 56,983 elements around Cape Fear, North Carolina. Once extracted, the subdomain and its boundary conditions then serve as a parent domain in an ASM simulation. To generate a local change for testing, we raise the topography in the Walden Creek area north of Southport, as shown in Fig. 2.25, which results in a 2.5-mile protective structure that prevents flooding in a region zoned for heavy industry and military.

The expansion and contraction of a child domain with $\tau^0 = 10^{-5}$, $\sigma = 10^{-2}$, $\lambda = 5 \times 10^{-4}$, and $\theta = 10$ is shown in Fig. 2.26. Since, in this case, local changes are made to dry nodes, the extent of the patch remains the same as its initial extent until timestep 539,602 (about 80% of simulated time). Once surge effects reach the locally modified region, however, the patch begins to expand to accommodate the altered hydrodynamics induced by the structure. As the storm surge retreats and the effects of the local change dissipate, the patch begins to contract.
Figure 2.23: Case 2: max-norms of maximum elevation errors in Hatteras child domains.

Figure 2.24: Case 2: Average percentage of active elements in Hatteras child domains.
For the parametric study, 144 child domains are concurrently simulated using all combinations of the following control parameter settings: $\tau^0 = \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}\}$, $\sigma = \{10^{-1}, 10^{-2}, 10^{-3}\}$, $\lambda = \{5 \times 10^{-3}, 5 \times 10^{-4}, 5 \times 10^{-5}, 5 \times 10^{-6}\}$, $\theta = \{10, 100\}$. Once again, to evaluate the influence of those settings, we perform a baseline run of the parent domain after the same local change has been made to it, and compute the $l^2$-norms and max-norms of maximum elevation errors of child domains, as shown in Figs. 2.27 and 2.28. The average percentage of active elements for each child domain is shown in Fig. 2.29.

Changes in initial tolerance and the decay constant have the most significant influences while the influences of changes in activation interval and the contraction factor are less significant. As the initial tolerance is reduced, the child domain patches expand further and accuracy improves. Similarly, as the decay constant is increased, the local tolerances converge to the initial tolerance more quickly, and so once again the accuracy improves. The maximum error in maximum elevations is 0.69 cm for the best combination of settings ($\tau^0 = 10^{-6}$, $\sigma = 10^{-3}$, $\lambda = 5 \times 10^{-3}$ and $\theta = 10$), 3.7 cm for the worst ($\tau^0 = 10^{-1}$, $\sigma = 10^{-1}$, $\lambda = 5 \times 10^{-6}$ and $\theta = 100$), and less than 1 cm for most of them.
Case 4: Brunswick intake canal and Hurricane Irene (2011)

For the final case, we again apply HSM in the context of a hurricane storm surge scenario, but this time using a more refined grid of the western North Atlantic: NC Mesh Version 9.98 with 622,946 nodes and 1,230,430 elements. Otherwise the extent and model parameters mostly correspond to those given in Case 3. For Hurricane Irene, a best-track file from the NOAA NHC online data archive is used for the meteorological forcing, and a 0.5-s step size is used to perform an 8-day simulation of the event as a 2DDI analysis. For this example, no tidal forcing is applied, thereby eliminating a long tidal spin-up run.

Working in the same area as before, we perform a full-scale run to obtain boundary conditions for a circular subdomain consisting of 39,234 nodes and 78,114 elements around Cape Fear, North Carolina. Our area of focus this time is the intake canal of the Brunswick nuclear power plant, where for a local change we increase the depths of all 244 nodes by 2 m and modify
Figure 2.27: Case 3: $l^2$-norms of maximum elevation errors in Walden Creek child domains.
Figure 2.28: Case 3: max-norms of maximum elevation errors in Walden Creek child domains.
Figure 2.29: Case 3: Average percentage of active elements in Walden Creek child domains (with $\lambda$, $\tau$ axes reversed).
the associated Manning’s $n$ values from 0.02 to 0.012, as shown in Fig. 3.10.

The expansion and contraction of the child domain with ASM control parameters $\tau^0 = 10^{-3}$, $\sigma = 10^{-1}$, $\lambda = 10^{-5}$, and $\theta = 100$ is shown in Fig. 3.11. The patch expands as the hurricane storm surge approaches, and it reaches its largest extent at timestep 1203758 (about 87% of simulated time). As the effects of the hurricane and the altered hydrodynamics dissipate, the patch contracts almost to its original extent.

For the parametric study, 96 child domains are concurrently simulated using all combinations of the following control parameter settings: $\tau^0 = \{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}\}$, $\sigma = \{10^{-1}, 10^{-2}, 10^{-3}\}$, $\lambda = \{1 \times 10^{-5}, 3 \times 10^{-6}, 1 \times 10^{-6}, 3 \times 10^{-7}\}$, $\theta = \{10, 100\}$. As in other cases, we perform a baseline run of the parent domain after the same local change has been made to it, and compute the $l^2$-norms and max-norms of maximum elevation errors of child domains, as shown in Figs. 2.32 and 2.33. The average percentage of active elements for each child domain is shown in Fig. 2.34.

The order of accuracy is the same as that of Case 3, though here the maximum errors are less than 3 mm for all combinations of control parameters, except for a child domain with
Figure 2.31: Case 4: Progression of a Brunswick child domain patch for $\tau^0 = 10^{-3}$, $\sigma = 10^{-1}$, $\lambda = 10^{-5}$, and $\theta = 100$, with elements in the patch darkened. Snapshots: (a) initial extent, (b) extent at timestep 1098072, (c) largest extent (occurring at timestep 1203758), (d) final extent (timestep 1374294).

$\tau^0 = 10^{-4}$, $\sigma = 10^{-2}$, $\lambda = 10^{-6}$, and $\theta = 10$, and a child domain with $\tau^0 = 10^{-4}$, $\sigma = 10^{-2}$, $\lambda = 3 \times 10^{-7}$, and $\theta = 10$. The largest errors of these two child domains, 4.7 cm and 2.2 cm, respectively, are observed to be occurring in wetting and drying regions. Given the nonlinear and discontinuous nature of the wetting and drying algorithm, such discrepancies are sometimes observed to occur, but in very isolated circumstances.

Discussion

The foregoing study on control parameters suggests that accuracy generally improves with (a) reductions in the initial tolerance and (b) increases in the decay constant, as one might expect, but there are caveats. For the initial tolerance, the largest improvements in accuracy seem to occur when it is reduced from $10^{-2}$ to $10^{-3}$ or $10^{-4}$. Further reductions, however, to say $10^{-5}$, improve accuracy only marginally while degrading the computational efficiency significantly and increasing the risk of patches reaching the grid boundary.
Figure 2.32: Case 4: $l^2$-norms of maximum elevation errors in Brunswick child domains.
Figure 2.33: Case 4: max-norms of maximum elevation errors in Brunswick child domains.
Figure 2.34: Case 4: Average percentage of active elements in Brunswick child domains (with \( \lambda, \tau \) axes reversed).
In addition to parameter settings, some modeling scenarios are inherently either more or less likely to produce early termination as a result of patches reaching a boundary. For instance, storm surge simulations focusing on overland flows and local changes in topography are usually more robust even for very low initial tolerances, as seen in the Walden Creek example with Hurricane Fran. In other cases, such as the Brunswick intake canal problem, local changes in bathymetry do influence the hydrodynamics early in the simulation, but less stringent tolerances still allow patches to expand and accommodate those influences appropriately, long before any surge effects come into play. As a result, the ASM technique can accurately simulate diverse modeling conditions, even when local changes occur near grid boundaries, as demonstrated here.

Based on these and other studies we have performed on the accuracy and efficiency of the method, a good balance seems to be found with ASM control parameter values of $\tau^0 = 10^{-3}$, $\sigma = 10^{-1}$, $\lambda = 10^{-4}$, and $\theta = 10$, so these constitute our default settings for ADCIRC++.

2.5.2 Applications and Performance

To further demonstrate subdomain modeling and its computational advantages, we look at the following additional cases:

5. Brunswick intake canal at Southport, NC

Hurricane Irene (2011) simulated as before on the more refined Cape Fear subdomain, but this time over a range of depths and Manning’s $n$ values along the intake canal

6. Silver Lake at Wilmington, NC

Hurricane Fran (1996) simulated on a small portion of the Cape Fear River, with varying values of Manning’s $n$ and depth on a part of the river bank

As before, each case is simulated with a number of concurrent child domains, but here the local changes are problem domain changes that are carried out using a constant set of ASM control parameter values. By again making use of conventional, static subdomains for the
parents, they also demonstrate the complementary benefits of subdomain modeling approaches realized by HSM.

**Case 5: Brunswick intake canal and Hurricane Irene (2011)**

Instead of control parameters, in this case we vary bottom surface conditions for a new set of child domains located at the same intake canal of the Brunswick nuclear power plant. Throughout its length, we use a constant Manning’s n value of either 0.012, 0.024, 0.048, or 0.096, which ranges from constructed channel conditions to ones that are unmaintained and have dense brush and weeds. Simultaneously, 17 different changes in depth, from \(-2\) m to 2 m, are made to the original bathymetry of the canal in the parent domain. With recording stations shown in Fig. 2.35, a simulation of the resulting 68 child domains is performed using the following ASM control parameter settings: \(\tau_0 = 10^{-4}\), \(\sigma = 10^{-2}\), \(\lambda = 10^{-5}\), and \(\theta = 100\).

![Figure 2.35: Case 5: Recording stations at the intake canal.](image)

Fig. 2.36 shows the water surface elevations at each recording station for each of the child domains. Changes in Manning’s n values have little effect except when canal depths are reduced to a point where wetting at the southwest end of the channel is prevented.

**Case 6: Silver Lake and Hurricane Fran (1996)**

Using the same grid from Case 3 encompassing the western North Atlantic Ocean, the Caribbean Ocean, and the Gulf of Mexico [12], we generate a small subdomain consisting of 11255 nodes...
As a hypothetical problem context, a development activity adjacent to the river seeks materials ecologically best suited to lowering water velocities during a hurricane event. To simulate a range of such materials, Manning’s n values are varied along two rows of nodes (50 nodes in total) in the region shown in the figure. The following Manning’s n values are considered: 0.015, 0.041, 0.067, 0.093, 0.119, 0.145, 0.172, 0.198, 0.224, and 0.250. Additionally, to simulate the effects of planned gabion walls of different sizes, the inner row of nodes closer to the river is adjusted in height ranging from 0 m to 0.5 m increases. With recording stations shown in Fig. 2.38, a simulation of the resulting 60 child domains is performed using the following ASM control parameter settings: $\tau^0 = 10^{-4}$, $\sigma = 10^{-1}$, $\lambda = 10^{-4}$, and $\theta = 100$.

Fig. 2.39 shows the velocities of all child domains at the recording stations. As indicated
by the plots, both roughness and raises in topography can be used, whether separately or in combination, to reduce water velocities during the hurricane event simulated.

**Performance**

The computational efficiency of ASM depends on numerous factors, including control parameter settings, model settings, and the spatial and temporal extent of the impacts of local changes. We evaluate the performance of the technique for the two cases in this section by comparing runtimes on a 64-core AMD Opteron Processor 6274 workstation using a serial prototype of ADCIRC++, an unoptimized pre-release version that nevertheless comes within about 15% of the (serial) performance of ADCIRC itself.

Tables 2.1 and 2.2 compare the computational costs of full-scale runs, CSM subdomains, and ASM child domains for the Brunswick intake canal and Silver Lake test cases. ASM child
Figure 2.39: Case 6: Velocities at recording stations for varying elevation raises and Manning’s n values.

Table 2.1: Comparison of computational costs for Case 5.

<table>
<thead>
<tr>
<th>Runtime</th>
<th>CPU hours</th>
<th>% of full scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full-scale grid</td>
<td>1488</td>
<td>100</td>
</tr>
<tr>
<td>CSM subdomain</td>
<td>102</td>
<td>6.88</td>
</tr>
<tr>
<td>ASM child domain</td>
<td>3.62</td>
<td>0.24</td>
</tr>
</tbody>
</table>

In both cases, the use of very small tolerances leads to patch sizes larger than necessary and correspondingly high accuracy, with errors less than a millimeter. Nevertheless, the runtime of a child domain is only a tiny fraction of a subdomain run, which itself is already a fraction of a full-scale run, so the combination constitutes a highly efficient use of resources. In terms of a cost breakdown for the components of ASM, it should be noted that little if any price is paid
Table 2.2: Comparison of computational costs for Case 6.

<table>
<thead>
<tr>
<th>Runtime</th>
<th>CPU hours</th>
<th>% of full scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full-scale grid</td>
<td>897</td>
<td>100</td>
</tr>
<tr>
<td>CSM subdomain</td>
<td>9.32</td>
<td>1.04</td>
</tr>
<tr>
<td>ASM child domain</td>
<td>0.19</td>
<td>0.021</td>
</tr>
</tbody>
</table>

for adaptivity since memory management is optimized and changes are infrequent (typically less than once every thousand timesteps on average). As a result, costs for ASM subdomains are largely proportional to their extent.

2.6 Conclusions and future work

Subdomain modeling techniques, whether adaptive or conventional, are designed to assess the effects of incremental changes at an incremental computational cost. Motivated by engineering design and failure scenarios, such techniques also support scientific studies where one or more local properties of a physical domain are varied over a meaningful range of values. Subdomain modeling is predicated on the observation that many changes of interest induce responses with a local extent and without producing effects far from their origins—at least at the space and time scales of interest. Thus, we can eliminate calculations that fall outside the sphere of influence of those changes.

The adaptive approach presented in this study offers new, attractive features that complement conventional subdomain modeling. By automatically adjusting boundaries in response to domain changes, ASM relieves users from determining the sizes and shapes of subdomain grids, provides greater performance gains, and eliminates the verification steps required by CSM. Error indicator settings and other control parameters determine the behavior of the algorithm, allowing users to tailor its accuracy and efficiency according to their needs. Most importantly, the overall computational approach, where parent and child domains are analyzed concurrently, imposes no arbitrary limitation on what is considered a parent, so users can employ conven-
tional subdomains as parents and do so hierarchically to any degree of nesting desired, giving rise to the combined HSM approach we describe.

The dynamic nature of ASM patches and other features, such as inter-domain communication, call for a software architecture that can accommodate them. We knew from the beginning we wanted to take advantage of ADCIRC’s mature and well-tested formulation because of its many modeling strengths, but we also knew it would require an overhaul of static data structures that were conceived under a different set of assumptions. After hand translating parts of the code with some success, we grew confident we could implement about eighty percent of its features and create an adaptive code that, while not highly optimized, could serve as a prototype for ADCIRC++ and a proof of concept for ASM.

More recent versions of ADCIRC++ incorporate inter- and intra-domain parallelism on multicore architectures, which is realized by a thread pool and a timestepping routine that allows each concurrent domain to be executed with one or more dedicated threads. A phasing mechanism prevents a parent domain from updating itself until children access the data they need and likewise prevents children from moving ahead of the parent domain. To minimize false sharing among threads, decomposition of a grid into multiple patches is performed by METIS, a graph partitioning library also used by ADCIRC. Ongoing efforts are focused on improving the performance of intra-domain parallelism on large numbers of processors.

With regard to other directions, we anticipate using ASM to facilitate new population-based optimization strategies that might result in next-generation decision-support systems for coastal engineers. More generally, we expect to continue our focus on tools and techniques for engineering users of large-scale storm surge models.

Acknowledgments

The authors thank Yoonhee Park for motivating the Silver Lake case study and offering the perspective of a landscape architect.

Team members of the Computational Modeling Group at NCSU provided helpful and con-
structive feedback during the course of this research, including Tristan Dyer, Fatima Bukhari, Yuliya Siamashka, Xing Liu, and Tucker Whitesides, and their contributions are appreciated.
Chapter 3

A Generic Software Architecture for Concurrent and Adaptive Ocean Models

Abstract  We introduce a generic software architecture for numerical ocean models, based on object oriented programming and data abstraction. The architecture is designed to facilitate adaptive behavior and to utilize concurrent and hierarchical executions of multiple domain instances. To demonstrate the applicability of the new architecture and to incorporate an adaptive technique, we re-implement ADCIRC ocean circulation model, originally based on global and non-reentrant data structures and procedural programming. In addition to adaptivity and concurrency capabilities, the architecture provides the common advantages of data abstraction over procedural programming such as modularity, maintainability and extensibility.

3.1 Introduction

Numerical ocean models are commonly used by scientists and engineers to assess the effects of coastal events, such as tides and hurricane storm surge, on the built and natural environment. Like most scientific software, ocean models are usually based on procedural abstraction, i.e., they are structured by dividing the control flow into reusable subroutines operating over global data structures. Although procedural programs written in imperative languages such as Fortran and C are recognized to be more efficient for simple data structures and operations, they lack
the mechanisms and ability to operate efficiently on complex data structures that can facilitate the implementations of next-generation approaches.

In this paper, we introduce a modern software architecture for numerical ocean models, based on object oriented programming (OOP) and data abstraction. Written in C++14, the architecture includes two levels of source code. The first level, called OpenHDM, is a generic framework consisting of class templates and abstract classes forming the structure of the architecture. This level is independent of implementation-specific details such as discretization, numerical methods, formulation, etc., and is common to all model types. The second level is implementation-specific source code, consisting of concrete classes derived from the generic framework, and specialized for the details of the model to be developed.

The architecture contains dynamic, reentrant and type-independent data structures that fully support the adaptive grid behavior. Discrete model data is maintained by a Grid/Patch pair, where the former is the container of the actual data e.g., nodes elements, etc., that is resized relatively less frequently in runtime, and the latter is a virtual view of the data, providing a level of indirection and allowing the designation of the active regions of the grid in a highly dynamic manner. Aside from dynamism, a unique feature of the data architecture is the support for multiple domain instances, since it was initially designed to accommodate the adaptive subdomain modeling (ASM) approach, a computational technique for the analysis of multiple design scenarios.

A built-in shared-memory parallelism implementation for the concurrent execution of multiple domain instances is included in OpenHDM, which uses the C++ standard threading library to eliminate any external library dependencies and to improve the portability. The concurrent execution of domains which may be defined in a hierarchy is synchronized through a phasing mechanism, where, at each timestep, the phase functions, each including a portion of the timestepping tasks, of domains are either executed or blocked depending on several criteria regarding the relative progressions of domains within the timestep. In addition to mechanisms for domain synchronization, this generic framework provides a thread pool implementation that
is responsible for regulating the executions of phases, depending on the number of concurrent
domains and available processors.

To demonstrate the applicability and efficiency of the architecture, we describe the develop-
ment of a simple finite-volume shallow water equations solver, called ASMFV, which is used
as a proof-of-concept for both the software architecture and the ASM approach. We then de-
scribe our re-implementation of ADCIRC, an advanced ocean circulation model, with the new
architecture and with the incorporation of the ASM approach. Originally based on procedural
programming with global and non-reentrant data structures, the new implementation, called
ADCIRC++, acquires the adaptive grid behavior, as well as the common advantages of object
oriented design.

In Section 4.3, we provide a background information on data abstraction, ASMFV for-
mulation, ADCIRC, and the ASM approach. In Section 3.3, we present a brief overview of
the previous studies on the application of modern design approaches for numerical modeling.
In Section 3.4, we describe OpenHDM, i.e., the first-level generic framework of the architec-
ture, which is followed by the descriptions of ASMFV and ADCIRC++ in Sections 3.5 and
3.6, respectively. We also present a case study to assess the computational performance of the
architecture.

3.2 Background

3.2.1 Data Abstraction and OOP

In software design, data abstraction is the separation of the implementation details of a data
type from its specification, i.e., its external view. Object oriented programming, which involves
data abstraction, is a programming paradigm based on structuring the code with respect to
classes that encapsulate data and functions as members, as opposed to procedural programming,
which is based on decomposing the code into subroutines operating over global data structures.

Data abstraction and encapsulation ensure that the changes made to the internal represen-
tation of a class do not propagate to the rest of the program, as long as the specification remains consistent. This abstract behavior, achieved by the representation independence, promotes modularity, enables the incremental development of a complex software system, and simplifies the verification and debugging procedures [13]. In addition to data abstraction and encapsulation, object oriented programming provides additional concepts improving the effectiveness of a software design. Polymorphism, for instance, allows the declarations and implementations of type-independent data structures and provides abstraction from the type-specific implementation details when designing general functionalities like grid adaptivity or domain hierarchy, while inheritance enhances code re-usability and type specializations.

Despite the well-established advantages of object oriented programming and data abstraction, the majority of the numerical software used in practice remains to be based on procedural abstraction. One of the reasons for procedural abstraction to remain favorable in numerical computing is that Fortran and other imperative languages are regarded to be more intuitive for science and engineering community [32]. Additionally, although the latest revisions of Fortran support object-oriented design, programming habits of scientists and engineers, and effort that would be required to transform existing programs appear to prevent the transition from procedural abstraction to data abstraction. Another reason is that object oriented languages are regarded to be computationally less efficient, with the exception of C++ which can perform as efficient as imperative languages unless a significant abstraction penalty is apparent due to high-level programming. Nevertheless, such abstraction penalty is insignificant when compared with performance gains obtained from the next-generation approaches like adaptive mesh refinement and adaptive subdomain modeling, which can be applied more effectively with OOP and data abstraction. Moreover, the computational complexity of a numerical program, hence the time spent on its development and debugging, is significantly reduced with the use of OOP and data abstraction [45, 86].

Consequently, our software architecture is based on object oriented design and is written in C++. A criticism for the use of C++ in scientific programming is its high programming
complexity resulting from the difficulty in dynamic memory management and exposure to low-level entities [85]. However, with the advent of modern C++, namely C++11 and C++14 which is used in this study, programming complexity has been reduced significantly by new tools and features such as automatic type deduction, reference wrappers, lambda functions, the new standard template library (STL) algorithms, and most importantly, smart pointers that relieve users from low-level memory management.

3.2.2 ASMFV Formulation

The formulation of ASMFV, a finite-volume shallow water equations solver that is developed with our generic architecture as described in Section 3.5, is based on the work of LeVeque et al. on the finite-volume methods for the shallow water equations [9, 71, 68, 70], and similar to that of GeoClaw [17]. ASMFV can perform both one-dimensional and two-dimensional simulations. The form of 1-dimensional shallow water equations solved by ASMFV is given as [71]:

\[ q_t + f(q)_x = \psi \]  \hspace{1cm} (3.1)

where, \( q \) is the vector of conserved quantities, \( f(q) \) is the flux function, and \( \psi \) is the vector of source terms, which are defined as:

\[
q = \begin{bmatrix} h \\ hu \end{bmatrix}, \quad f(q) = \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \end{bmatrix}, \quad \psi = \begin{bmatrix} 0 \\ -ghB_x \end{bmatrix}
\]

where, \( h \) is the water column height, \( u \) is the velocity in x direction, \( g \) is the gravitational acceleration, and \( B \) is the bathymetric depth.

For the finite-volume solution of the shallow water equations, we use the f-wave method, a reformulation of wave-propagation form of Godunov method, that has practical advantages for the incorporation of source terms such as the variable bathymetry [71]. Both the wave-
propagation and f-wave methods are based on defining a Riemann problem at interfaces between adjacent cells with discontinuous solutions. In wave-propagation form, the Riemann problem is specified to compute the waves corresponding to jumps in conserved quantities, namely mass and momentum, across characteristic curves [68]. In f-wave form, however, the waves (denoted by $Z^p$ for the p.th component) are computed as the jumps in fluxes instead [71]:

$$
\sum_{p=1}^{\mathcal{M}_w} Z^p_{i-1/2} = f(Q_i) - f(Q_{i-1})
$$

(3.2)

where,

$M_w$ : number of characteristic waves

$f(Q_i)$ : flux at i.th cell

$\beta = R^{-1}(f(Q_i) - f(Q_{i-1}) - \Delta x \psi_{i-1/2})$

$R$ = matrix of eigenvectors (invertible)

$\psi_{i-1/2}$ : discretization of source terms at interface between cells $i$ and $i - 1$

$r^p$ : p.th eigenvector of the system

Once the waves are computed, their combined net effects at each cell are calculated to obtain the numerical solutions at the next timestep. The numerical form based on the second order f-wave formulation is given as [9]:

$$
Q^n_{i+1} = Q^n_i - \frac{\Delta t}{\Delta x} \left[ \sum_{p: s^p_{i-1/2} > 0} Z^p_{i-1/2} + \sum_{p: s^p_{i+1/2} < 0} Z^p_{i+1/2} \right] \\
- \frac{\Delta t}{2\Delta x} \left[ \sum_{p=1}^{m} sgn(s^p_{i+1/2})(1 - \frac{\Delta t}{\Delta x} |s^p_{i+1/2}|) \tilde{Z}^p_{i+1/2} \right] \\
- \sum_{p=1}^{m} sgn(s^p_{i-1/2})(1 - \frac{\Delta t}{\Delta x} |s^p_{i-1/2}|) \tilde{Z}^p_{i-1/2} 
$$

(3.3)
where,

\[ Q_i^n : \text{vector of conserved quantities at timestep } n \text{ and cell } i \]

\[ \Delta x : \text{length of a cell} \]

\[ \Delta t : \text{duration of a timestep} \]

\[ s_{i-1/2} : p^{th} \text{ characteristic wave speed at interface between cells } i \text{ and } i - 1 \]

\[ \tilde{Z}^p : \text{limited f-wave for the } p^{th} \text{ component} \]

\subsection{3.2.3 ADCIRC Background}

ADCIRC is an unstructured-grid finite-element ocean circulation model used by scientists and engineers to simulate the effects of tides and hurricane storm surge in coastal regions. Spatial discretization in ADCIRC is based on a continuous Galerkin formulation, while time discretization may be configured as a variably weighted three-time-level implicit scheme or an explicit scheme [74].

At every timestep, ADCIRC solves the Generalized Wave Continuity Equation (GWCE) to compute the water surface elevations. The formulation of the GWCE eliminates the spurious oscillations resulting from the finite-element discretization of the shallow water equations [76]. Following the solution of the GWCE, ADCIRC executes the wetting and drying algorithm to determine the wet/dry states of nodes and elements depending on several inundation criteria, and finally, solves the momentum equations to obtain the velocities and fluxes at every grid node.

ADCIRC simulations can be performed in serial or in parallel. In case of parallel execution, a preprocessor decomposes the computational grid into subdomains with approximately equal numbers of nodes and elements, where each subdomain is executed by a dedicated processor [12]. In the parallel version, the exchange of nodal data, including the water surface elevations,
ADuCIRC is tightly coupled with SWAN, a third-generation spectral wave model [22, 38]. External libraries used by ADuCIRC include ITPACKV [63] for the iterative solution of the sparse linear GWCE system, KDTREE-2 [62] for element selection, NetCDF [84] as an output format alternative, METIS [61] for the domain decomposition in the parallel version, and MPI as the communication interface between processors.

### 3.2.4 Adaptive Subdomain Modeling

Adaptive subdomain modeling is a multi-analysis technique for numerical ocean models to assess the effects of multiple local changes applied to an original model configuration [3]. The technique is realized by concurrent executions of an original full-scale domain, called parent domain, and any number of locally modified domains, called child domains, corresponding to various engineering design and failure scenarios. Initially covering only the regions with alterations, the spatial extent of child domains are adaptively adjusted during runtime, where an assessment is made at the beginning of every timestep around the child domain boundaries to determine the proximity of altered hydrodynamics, i.e., the differences in solutions of an original parent domain and a modified child domain, originating from the local alterations. If it is determined that the altered hydrodynamics can potentially reach the boundary of a child domain within a timestep, and so invalidate the boundary conditions obtained from the parent domain, the boundary is locally moved outwards. If it is determined that the altered hydrodynamics is far enough from the boundary, on the other hand, the boundary is moved inwards to reduce the computational effort. Boundary conditions for the moving boundaries of child domains are obtained from the concurrent parent domain and transferred on-the-fly without a need for secondary memory usage. The method is implemented both in ASMFV and ADuCIRC++. Test results show that the application of the method leads to a highly efficient and accurate reanalysis of multiple local changes.

In adaptive subdomain modeling, the decision of whether a child domain patch grows,
shrinks, or stays the same is made via assessment of an error indicator ($\rho$), which is a measure of the difference between the solutions of a child domain and its parent domain, around the patch boundary. If the error indicator is determined to be greater than a tolerance ($\tau$) set by the user, the patch boundary is moved outwards. If the error indicator is sufficiently small, then the boundary is moved inwards.

Since the method requires the coexistence and interoperability of multiple hierarchical domain instances, object oriented programming concepts utilized in our software architecture, such as data abstraction and encapsulation, greatly enhance the applicability of adaptive subdomain modeling. Additionally, dynamic data containers and the Grid/Patch pair used in the architecture enable the adaptive grid behavior i.e., the expansion and contraction of child domains, to be implemented efficiently. As an example, Figure 3.1 shows the expansion and contraction of an ADCIRC++ child domain patch encompassing only the regions of altered hydrodynamics and designating the nodes and elements that participate in the computations.

Figure 3.1: Expansion and contraction of locally modified Shinnecock Inlet child domain patch at various timesteps. (Elements included in the patch are highlighted in darker colors.) [3]
3.3 Related Work

Although procedural programming is still favored by the numerical modeling community, there is a considerable amount of effort towards the utilization of modern design approaches for scientific software development. In this section, we briefly review several libraries, architectures and frameworks developed for high-performance numerical modeling that are based on design patterns such as component-based architecture and object oriented programming.

The Common Component Architecture (CCA), a development environment for scientific models, is a component-based software architecture for multidisciplinary simulations where distinct model components can interact by means of an interface definition language that supports scientific abstraction and a ports communication model for collective interactions [7]. The interface definition language of CCA is described by Armstrong et al. as a high level description language for specifying the interfaces of components, which may potentially be developed in different programming languages, while ports, another essential element of the standard, are defined as end points that are used to implement communication links between components [7]. Thus, the tools and mechanisms of CCA, which are the products of component-based architecture design, facilitate interoperability and compatibility.

The Earth System Modeling Framework (ESMF) similarly specifies an integrated multi-component modeling environment for Earth science applications [53]. Developed to provide interoperability for previously incompatible models, ESMF’s essential components are listed by Hill et al. as the user code that models the scientific elements, the ESMF superstructure encompassing the user code and interconnecting the model components, and the ESMF infrastructure that provides a support library used to ensure the consistency of components. Implemented in a combination of Fortran and C++, ESMF is based on object-oriented programming and component-based architecture [30]. Both the superstructure and infrastructure of ESMF is organized as a collection of classes that provide flexibility and compatibility to facilitate component interoperability [30, 53].
The libMesh, an open-source library for parallel, adaptive, multiphysics simulations is another notable example of the utilization of advanced programming paradigms in numerical modeling [64]. Designed to facilitate adaptive mesh refinement for unstructured grids, the library aims to hide the implementation details of parallelism for physics modelers [64]. Written in C++, the library makes use of both run-time and compile-time polymorphism through abstract classes and templates, and provides a suite of data structures to be used for numerical model development. Kirk et al. note that the library was written in C++, instead of a lower-level procedural language, to obtain both efficiency and extensibility, at the expense of inter-language operability, and point out that C++ and object oriented design make it possible to improve maintainability and reduce complexity introduced by adaptivity and parallelism [64].

OpenFOAM is an open source, object oriented modeling library that has a variety of application areas including computational fluid dynamics and structural mechanics [59]. Jasak et al. [60] argue that monolithic functional programming with global data structures lead to extensibility issues and complicate the validation efforts, whereas object oriented design leads to efficient and versatile programs owing to the techniques such as data encapsulation, operator overloading and polymorphism. Therefore, the library was written in C++ and has been designed to exploit object oriented programming principles. They also note that high-level programming languages like C++ make it more convenient to optimize a generic code since changes applied for optimization do not propagate beyond the low-level objects, and so do not affect the users of the generic classes [60].

Rouson and Rosenberg present a grid-free abstraction, i.e., a continuous representation, of Navier-Stokes Equations, and argue that object-oriented design utilizes code-reuse, which shortens the development time through inheritance, composition, and aggregation, while information hiding simplifies debugging process [86]. They implement their object oriented design in Fortran, and point out that global sharing of data in structured programming, as opposed to object oriented programming, leads to propagation of a local error to the entire program and greatly increases the complexity of the code [86].
Other examples of object oriented libraries and architectures for physical modeling include COOLFluid [67], a C++ library for fluid dynamics simulations, Overture [24], an object-oriented environment for parallel and serial computations of PDEs using finite difference or finite volume methods in 2D and 3D, OpenGeoSys [65], an open-source numerical framework for finite-element simulations of thermo-hydro-mechanical-chemical processes, Tarsier [93], an environmental simulation modeling framework, etc.

3.4 The Architecture

Our software architecture consists of two levels of source code: (1) the generic OpenHDM framework, which is a collection of class templates and abstract classes forming the structure of the code, (2) an implementation-specific source code. The generic framework is common for all models based on our software architecture and requires no modifications. The implementation-specific source code is developed by the user and consists of concrete classes that are derived from OpenHDM and specialized for the model details to be developed.

Abstract classes and class templates of OpenHDM, which are described in Section 3.4.1, provide implementations for general functionalities such as the management of multiple hierarchical domains, concurrent and synchronized execution of the timestepping routine, adaptive tasks for discretized grids, I/O operations, reporting of logs, warnings and errors, etc. The generic data structures defined in the framework supports multiple domain instances and dynamic grid behavior and provides great flexibility for the implementations of ocean models with any type of numerical method and formulation.

Similar to the source code itself, the architecture prescribes the parallelism in two levels. The first level parallelism, which determines the inter-domain concurrency, is simple but robust, fully implemented in OpenHDM using the standard multithreading library of C++14, and requires no user modifications. Depending on the progression of hierarchical domains, as well as the availability of processors, the Project class template, executes and synchronizes the timestepping threads of domains using the data structures defined in Domain class and
**Threading** namespace, as described in Section 3.4.2. The implementation of domain-level parallelism, however, is not provided in the generic framework since it highly depends on the details specific to the model to be developed, and therefore left to the user.

We demonstrate the applicability of the architecture with two implementations, ASMFV and ADCIRC++, based on different formulations and numerical methods. ASMFV, described in Section 3.5, is an open-source finite-volume shallow water equations solver developed to be used for proof-of-concept experiments of ASM approach. ADCIRC++, described in Section 3.6, is the re-implementation of ADCIRC, an advanced ocean circulation model, with the incorporation of ASM capability.

### 3.4.1 OpenHDM

Before describing each data type in detail, we outline the architecture as follows. The class template **Project**, which can have multiple **Domain** instances, is the main driver of concurrent simulations. The constructor of **Project** expects a **ProjectInput** object as a parameter which encapsulates the information read from a project input file, including the list of domains to be instantiated and domain hierarchy information. A **Domain** object corresponds to an individual model instance that has a distinct set of inputs, a computational grid, and outputs. Each **Domain** instance has exactly one **Solver** instance including numerical libraries and implementations of computational methods. Each **Solver** instance has exactly one **Grid** instance containing the discrete model data. A **Grid** instance stores the **Unit** instances corresponding to discrete mesh components such as nodes, and elements, and can contain one or more **Patch** instances designating the active **Unit** instances. The simplified class diagram of the generic architecture is shown in Figure 3.2.

Both **Grid** and **Patch** are variadic class templates, i.e., they can be instantiated with arbitrary numbers of template parameters, which designate the type of discrete mesh components to be maintained. This provides flexibility to construct a variety of ocean models based on entirely different numerical methods and formulations. A finite-difference model, for instance,
may be constructed with a single template parameter, \texttt{Node}. A continuous Galerkin model, on the other hand, can have two component types to maintain, \texttt{Node} and \texttt{Element}, whereas a discontinuous Galerkin model may be developed with three template parameters, \texttt{Node}, \texttt{Element} and \texttt{Interface}.

In the remainder of this section, we describe the main data structures of OpenHDM. (The classes of the \texttt{Threading} namespace, and so the details of domain concurrency, are described in Section 3.4.2.)

\textbf{Project}

\texttt{Project} is a class template with a single template parameter determining the type of the domains to be simulated. The implementation of the class template is complete and requires no user modification or specialization. For each concurrent execution, a single \texttt{Project} instance is created, which is responsible for coordinating the domains and the timestepping routine. Once the \texttt{Project} object is instantiated, it constructs the domains listed in the project input file.
int main(int argc, char* argv[]){
    // Get the projectFilePath and no. of processors from CLI
    auto args = getArguments(argc, argv);

    // Read in the project input file:
    ProjectInput projectInput(args.first);

    // Construct the project instance:
    Project<ADCDomain> project(projectInput);

    // Run the project:
    project.run(args.second);

    return 0;
}

Figure 3.3: Main function of ADCIRC++

depending on the hierarchy defined in the same input file. The execution of a simulation begins when the member function run() is called, which initializes and configures the domains and their threads, and then performs the concurrent execution of the timestepping routine. As an example, Figure 3.3 shows the main function of ADCIRC++, where the instantiation of the project object is followed by a call to its member function run().

Domain

The abstract class template Domain is used as a base class for derived domain types declared in the implementation-specific source code, and is used to encapsulate the data of each individual domain instance. Encapsulated data objects include input and output files, a Solver instance, a Grid instance (through the Solver object), parameters, etc. The abstract class template leaves the implementation of tasks that fully depend on the model details, such as reading input files and lazy initialization of derived-type members, to the user through pure virtual functions. General tasks regarding domain hierarchy, however, are implemented in the abstract template, where each parent domain retains a vector of weak pointers to its children, and each child
domain retains a shared pointer to its parent. The abstract template also includes member
variables to be used for concurrency, including mutex objects, condition variables, instances of
classes declared in Threading namespace, as well as a vector of functions, named phases, which
is used to store the phase functions that constitute the consecutive stages of a single timestep.
The phasing mechanism of the architecture is vital for the synchronization of concurrent and
hierarchical domains and is further explained in Section 3.4.2.

Solver

The abstract class template Solver is used as a base class for derived solver types that im-
plement the computational tasks specific to the model and include the numerical libraries to
be used. Each solver has a grid object as a member which is used to store the discrete model
data. Solvers of child domains additionally own a shared pointer to the solvers of their parent
domains, which is used for data transfer.

Grid

The variadic abstract class template Grid is the container and organizer of discrete model data
for individual domain instances. As mentioned earlier, a specialization of Grid can have an
arbitrary number of unit types, such as nodes, elements, cells, etc., depending on the numerical
model. All types of units are stored in the same data member called unitsTuple, which is a
tuple that includes a separate vector for each unit type. Each grid instance can have multiple
patch instances designating its active units. The data members of Grid template fully supports
the adaptive behavior and its member functions provide the mechanisms for adaptive grid
behavior, which requires very minimal additions when a derived grid type is generated.

Patch

The variadic abstract class template Patch is used as a base class for specialized patch types
which are used to designate the active regions of grids, i.e, the regions at which the numerical
computations are to be carried out. **Patch** is designed to maintain pointers (as references) to the objects stored in **Grid**, and so similar to grids, patches can have an arbitrary number of unit types. Pointers to active units are stored in a member container of **Patch**, named **unitptrsTuple**, which is of type tuple that contains a separate vector for each unit type. Like **Grid**, the abstract template class **Patch** provides implementations for inserting and removing units. Compared to the **Grids**, **Patches** are designed to be more light-weight and more dynamic. For instance, a **Unit** of a child domain in an ASM implementation which may need to be activated and deactivated repeatedly during runtime can only be added once to a **Grid** when it is first needed and never removed, but may be added to and removed from a patch as many times as required. Introducing the adaptivity in this manner eliminates the computational cost of instantiating and destructing the same object repeatedly, and so improves the efficiency of adaptive techniques. Along with the pointers to active units, specialized patch classes can include other containers, which are to be sized according to the number of active units, such as coefficient matrices and load vectors of a linear system of PDE.

**Unit**

The abstract class **Unit** is used as a base class for data structures encapsulating the discrete mesh data. Example derived **Unit** types are nodes, elements, cells, interfaces, etc. Units have three different identifiers, First, **id**, which is of type integer, is used for i/o operations. Second, **pos**, which is of type unsigned integer, is used to designate the position of the unit within the grid. Finally, **patchPos**, which is also of type unsigned integer, is used to designate the position of the unit within the patch it is included. Compared to **pos**, the identifier **patchPos** is more dynamic and subject to frequent changes. Therefore, generally, **pos** is used to refer to and retrieve the units in the source code.
3.4.2 Inter-Domain Concurrency

Concurrent timestepping of multiple hierarchical domains is implemented in the generic framework, which makes use of the data structures of C++14 standard multithreading library, in addition to data types, Pool and ControlPoint, defined in the Threading namespace. During the initializations, Project instructs the parent domain to instantiate its threading objects including mutexes, condition variables, and a Threading::Pool object, and then provides child domains with shared pointers to these objects. Thus, each parent domain owns a Threading::Pool object and monitors that are cooperatively used with child domains for synchronization. Both parent and child domains own Threading::ControlPoint objects to designate the timestepping phase in which they are and to notify the connected domain(s), that is the parent domain for a child domain and vice versa.

The vital component of the domain concurrency is a synchronization mechanism implemented in Domain and realized by Project. Based on dividing a timestep into phases in accordance with the tasks and properties being modified, the mechanism prevents the parent domain from overwriting a property before it is copied by the child domains, and prevents the child domains from moving ahead of their parents. The phases, which are simply lambda functions with calls to the member functions of the domain and its solver instance, are defined and inserted into phases vectors of Domains during initializations, and are looped over and called during the simulation depending on the availability of processors and relative progress of the domains within the timestep.

Threads of each domain are initialized to execute a timestepping loop, where, at each step, phase functions stored in phases vector are consecutively called after several criteria are evaluated: Before the \((i + 1)\).th phase function of a parent domain is called from its execution thread, it is checked whether all of its children have completed \((np + i - 1)\%np\).th phase, where \% is the modulo operator, \(i = \{0..np - 1\}\), and \(np\) is the number of phases. This ensures that a parent domain can at most be one phase ahead of their children. If the criterion is met, the parent notifies their children and requests permission from its thread pool. If the criterion is not met,
or if there are not enough processors, the thread of the parent domain blocks until it is notified. Once a parent function completes the execution of a phase function, it updates the state of its `Threading::Pool` and `Threading::ControlPoint` objects, and notifies its child domains. The child domains can request permission from the thread pool of their parent to move on to the next phase only if their parent has completed that phase at that timestep. Once a child domain completes a phase function, it releases the processor obtained from the thread pool and notifies its parent. Figure 3.4 summarizes the criteria regarding the progression of related domains for parent and child domains to execute their phases.

![Diagram](image)

Figure 3.4: Phase execution criteria for parents and children

When a parent domain executes its $p$th phase at a timestep, a child domain may be in one of the following three states: requesting permission for the $(p-1)$th phase, executing the $(p-1)$th phase, or waiting for its parent domain if it has already completed the $(p-1)$th phase. Similarly, when a child domain executes its $p$th phase, the parent domain may be in one of the following three states: checking for the $(p+1)$th phase, executing the $(p+1)$th phase, or waiting for its child domains if it has already completed the $(p+1)$th phase. Figure 3.5 shows the state diagram, drawn by the LTSA model checking tool, of the composition of a concurrent parent domain and a child domain execution with two phases where the actions labeled with `ready`
and \textbf{run} represent a domain (designated by the prefix of the action) passing the criterion for a phase and executing the phase, respectively.

![State diagram of the composition of a parent domain and a child domain with two phases.](image)

Figure 3.5: State diagram of the composition of a parent domain and a child domain with two phases. (Actions with prefixes pr and ch correspond to the ones engaged by the parent and the child, respectively. The indices shown between square brackets show the phase index for the specified domain. The action \textit{notify} which follows the action \textit{run} is subjected to hiding operator for brevity.)

One by-product of the phasing mechanism is the ability for child domains to copy a property, e.g., a boundary condition component, from their parent domains directly in case the property to be copied is not modified by the parent domain during the phase following the phase at which the property is copied by the children. (The property may be modified by the parent domain at any other phase.) This condition is due to a potential overwrite of the value to be copied before or during its sampling by the children. The condition may become too restrictive in some cases due to performance reasons or due to modifications of multiple properties within a single phase. In that case, the restriction may be lifted using a secondary data structure for the
parent domain to record the value of the property at a phase, so that the actual variable may
be modified at the next phase without affecting the values to be copied by the child domains.

As an example, Figure 3.6 shows the flowchart of the executions of ADCIRC++ timestep
phases.

Figure 3.6: Flowchart of the timestepping loop of ADCIRC++ domains.

3.5 ASMFV

ASMFV is a simple finite-volume shallow water equations solver developed to be used for proof-
of-concept experiments of the adaptive subdomain modeling approach. As described in Section
3.2.2, the finite-volume solution of the shallow water equations in ASMFV is based on the
f-wave method [71]. The model is implemented with the generic architecture presented in this
study, and so, supports the execution of multiple hierarchical domain instances and adaptive
grid behavior. The code is open-source and is available from an online repository [1].

The implementation-specific source code of ASMFV consists of concrete classed derived
from the generic framework. The class SFVDomain, derived from the abstract Domain template,
encapsulates model data for each domain instance and includes implementations for the initial-
izations of the solver instance, handling of I/O objects, and insertion of timestepping phases.
For 1-dimensional ASMFV domains, the timestepping phases are as follows.
Phase 1: Patch adjustments and initializations

Phase 2: Riemann solver (x-sweep)

Phase 3: Computation of the numerical form of the conservation law (x-direction)

Phase 4: Post-processing

Each SFVDomain class owns an instance of SFVSolver, which is a class derived from the abstract Solver template and includes implementations for the computational routines. SFVGrid, the data container for discrete ASMFV grids, is derived from the variadic abstract class template Grid, with the patch type SFVPatch and a single unit type, Cell, as the template parameters. Cell encapsulates the properties of a single cell within a grid, including the average water column height, momentums, and characteristic wave properties. The Riemann solver and the computation of the numerical form of the conservation law are implemented as member functions of Cell, and are called by SFVSolver for each active cell instance during the timestepping routine. Encapsulating and hiding the computational properties of cells, e.g., conserved quantities and characteristic wave variables, ensures that any change made in the implementation of Cell does not propagate outside the class since the rest of the code only has access to its abstract representation.

The adaptive subdomain modeling implementation in ASMFV exploits the phasing mechanism for the transfer of boundary conditions from parent domains to child domains during runtime. During the implementation of the method, an ASMFV child domain requires two sets of properties from its parent. First, the values of the conserved quantities of the corresponding parent cells are compared with the values of the child domain cells near the boundary to assess the proximity of the altered hydrodynamics to the boundary. Second, the characteristic wave properties from the parent domain are required to enforce the outer interfaces of the boundary cells of child domains. For 1-dimensional domains, the conserved quantities of parent domains are sampled at the first phase, where the assessment of altered hydrodynamics is carried out to determine the adjustments to be applied to the child domains. When child domains execute
their first phase, parent domains can only execute their second phase, and since these quantities are not modified in the second phase, child domains can directly use the values of parent quantities. Similarly, the second set of properties, which is the characteristic wave properties, can directly be copied by the child domain in the third phase as boundary conditions, since they are not overwritten by the parent domain in the fourth phase.

3.6 ADCIRC++

The core capabilities of ADCIRC ocean circulation model is reimplemented with the new architecture to facilitate the implementation of adaptive subdomain modeling. The new model, called ADCIRC++, utilizes concurrent execution of multiple domain instances and accommodates adaptive grid behavior. To transform the data structure and the design pattern in compliance with the new architecture, over 25,000 lines of Fortran code written according to procedural programming principles is rewritten and tested extensively. In addition to the hand-translated code, the Jacobi conjugate gradient solver of ITPACKV is translated using Fable [49], an automatic Fortran to C++ conversion tool, and KDTREE-2, a library used in original ADCIRC for element selection [62], is replaced with its C++ version.

Reimplemented capabilities include cold-starting routine, 2DDI consistent and lumped mass matrix assemblers and solvers for GWCE, wetting and drying routine, non-conservative momentum equations solver, tidal and meteorological forcing routines, etc. Capabilities not reimplemented yet include SWAN coupling, NetCDF file formats, hot-starting, 3D solver, etc.

We make use of the Grid/Patch concept of OpenHDM to parallelize ADCIRC++: At the beginning of each simulation, large-scale grids are decomposed into multiple patches whose extents are determined by the METIS graph partitioning library. During the simulation, the computations of each patch are carried out by a dedicated OpenMP thread.

Figure 3.7 shows the class diagram of ADCIRC++, which includes the classes derived from the generic framework, as well as classes specific to ADCIRC implementation, e.g., NodalAttr that includes the data structures and functions for the special nodal attributes and Wind that
includes the implementation specifications for wind forcing inputs and computations. We first review the classes of ADCIRC++ that are derived from OpenHDM. Then, we describe the incorporation of adaptive subdomain modeling into ADCIRC++ and evaluate its performance with a case study. Finally, we discuss the advantages of encapsulation and data abstraction brought to the code in terms of extensibility and maintainability.

### 3.6.1 Derived classes of ADCIRC++

**ADCDomain**

**ADCDomain**, which is derived from the abstract class template **Domain**, encapsulates the data of each individual ADCIRC++ domain. Along with the inherited implementations regarding domain hierarchy, concurrency, and the phasing mechanism, the class includes additional implementations for pure virtual functions regarding inputs and member initializations. The insertion of phase functions, which are called by the **Project<ADCDomain>** object, is done in the constructor operator of **ADCDomain** by calling the **insertPhase** member function of the base class.
ADCSolver

ADCSolver, derived from the abstract class template Solver, includes implementations for numerical and computational tasks. The implementations provided by ADCSolver include the GWCE solver, wetting and drying scheme, momentum equations solver, etc, that are called by the phase functions of ADCDomain. The has-a relationship between solvers and grids provides an efficient access interface to discrete model data from ADCSolver instances. Since the implementation of solvers heavily depends on the model details, the inheritance from the base class is very minimal. Indeed, the functions in Solver class template, except the constructors and attribute accessors, are defined to be pure virtual functions. Figure 3.8 shows the member functions of Solver template and the derived ADCSolver class.

ADCGrid

ADCGrid class encapsulating the discrete model data of ADCIRC++ domains is derived from the specialization of Grid template where ADCPatch is set as the patch type, and Node and Element are set as the unit types. (Recall that a variable number of units may be specified for a Grid template specialization.) The member functions of grids, including the ones facilitating adaptivity, are generalizable and can be implemented without knowing the implementation details of concrete unit classes. Therefore, the majority of the member functions are implemented at the generic framework by means of compile-time polymorphism and directly inherited by...
ADCGrid without being overridden (except initializePatches() which is a pure virtual function), as seen in Figure 3.9.

**ADCPatch**

ADCPatch, derived from the specialization of Patch template with parameters Node and Element as unit types, inherits the container unitptrsTuple that stores the vectors of pointers to active nodes and elements. A grid instance in ADCIRC++ can have multiple patch instances. Unlike the original ADCIRC, where each domain instance has one linear system of PDE, an ADCIRC++ domain has as many linear systems of PDEs as the number of patches, and so each patch instance retains its own containers to be used in the matrix computations, e.g., the coefficient matrix (both the consistent and lumped mass matrices). ADCPatch also contains vectors that store the positions of boundary nodes, which need to be monitored and altered for the implementation of adaptive subdomain modeling.

**Node and Element**

In original ADCIRC, nodal properties, which are over a hundred in total, are stored in individual global arrays. In ADCIRC++, the Node class derived from the abstract class Unit encapsulates all of the nodal properties and separates the implementation details of a Node instance from its
representation in the remainder of the code. Similarly, **Element** class derived from the abstract class **Unit** encapsulates the elemental properties and provides representation independence. Since both the nodes and elements in ADCIRC++ inherit the identification style of **Unit**, they have a member variable named **id** that designates their number set in the grid input file and used in output files. Another member variable named **pos** designates the order of the nodes and elements in which they are inserted in **ADCGrid**, and **patchPos** designates their order in **ADCPatch**, and thus, the order in the computations.

### 3.6.2 Application of ASM in ADCIRC++

The ASM approach, which eliminates the requirement for users to determine the sizes and shapes of locally modified domains, is implemented in ADCIRC++ effectively, as a result of its software architecture that facilitates the concurrent execution of hierarchical domain instances and adaptive grid behavior through paired **Grid/Patch** system.

During the initializations, a parent domain reads in its conventional ADCIRC input files such as the grid input file, model configuration file, nodal attributes file, and meteorological forcing files. Once the inputs are read, the **ADCDomain** instance configures its members including **ADCSolver** and **ADCGrid**. For each instance of child domains, the only required input file is **.dif** file including a list of modified nodes and the new values of the modified properties, which are typically the bathymetric depth, bottom friction, sloping limiter, etc. Once a child domain reads the **.dif** file, it initializes and configures its members using the data transferred from its parent.

The patches of a parent domain in ASM implementation of ADCIRC++ encompass the entire grid. For a child domain, however, both the grid and the patches initially include only the nodes and elements listed in the **.dif** file, in addition to several layers of nodes and elements surrounding them. During the runtime, as the effects of the modifications begin to propagate to the outer layers of nodes and elements, both the grid and the patch of the child domain expands to ensure that the altered hydrodynamics stays within the extent of the child domain. The first time a node or an element needs to be activated for a child domain, it is emplaced in
the grid, and its pointer is added to the patch. When it is deactivated, its pointer is removed from the patch, but its actual instance is kept in the grid, so that when it needs to be activated again, its pointer can simply and efficiently be re-inserted into the patch.

To calculate the error indicator values around the boundary, child domains get the values of the parent domain solutions in the first phase, where the necessary patch adjustments are evaluated and applied. The boundary conditions for the child domain patch boundary nodes are directly transferred from concurrent parent domain through primary memory, safely, thanks to the phasing mechanism. The water surface elevations, that is the first component of the boundary conditions, is transferred at the third phase of each timestep, which is the phase the elevations are iteratively computed both in parent domains and in child domains. The remaining boundary conditions, wet/dry states and velocities, are transferred at the fourth phase where the wetting and drying algorithm and the momentum equation solver are consecutively executed.

### 3.6.3 A case study: Hurricane Irene (2011) storm surge

To assess the performance of ASM and ADCIRC++, we present a sensitivity analysis, based on a previous case study where we combined the CSM and ASM approaches to analyze Hurricane Irene storm surge at the intake canal of a nuclear power plant near Cape Fear, NC [3].

To simulate various bottom surface conditions for the intake canal, we generate 68 different child domain instances where we vary the Manning’s N values (0.012, 0.024, 0.048 or 0.096) and the bathymetric depths (with decrements and increments from -2m to 2m to the original depths) of the nodes within the canal. We set a circular subdomain with 78,118 elements (Figure 3.10), previously extracted from a full-scale grid consisting of 1,230,430 elements and encompassing the western North Atlantic, as the concurrent parent domain of the child domains. In the previous study, we set the initial tolerance for the error indicator to $10^{-4}$, a very small value that leads to a highly sensitive multi-analysis of the scenarios and very large child domain patches. In this study, we set the tolerance to $10^{-3}$, which reduces the sizes of patches and improves the efficiency without significantly reducing the order of accuracy. (The maximum water elevation
errors are less than a millimeter for both tolerance values).

We carry out our ADCIRC++ runs on a 64-core AMD Opteron Processor 6274 workstation, and evaluate the performance of ASM and ADCIRC++ by comparing the runtimes. As noted in the original study [3], the conventional subdomain modeling approach reduces the computational cost of an additional simulation from 1487.60 CPU-hours to 102.35 CPU-hours, for the concurrent parent domain. The ASM approach with $\tau = 10^{-4}$ further reduces the cost of an additional simulation to 3.62 CPU hours, i.e., 0.24% of the full-scale run. The same concurrent ASM run with $\tau = 10^{-3}$ reduces the cost of an additional simulation to 1.02 CPU-hours, which is 0.07% of the full-scale run.

3.6.4 Discussion

The incorporation of ASM in ADCIRC++ is an example of how data abstraction and other object oriented programming concepts help design numerical software architectures that can effectively accommodate adaptive techniques. Although it may cause slight efficiency losses for simple operations and data structures due to abstraction penalty, the gain from high-level programming approaches is the ability to utilize computational techniques, that in return, provide overwhelmingly more efficiency. For conventional ADCIRC runs, for instance, the computa-
Figure 3.11: Progression of Irene child domain patch for $\tau = 10^{-3}$, $\sigma = 10^{-1}$, $\lambda=10^{-5}$, $\theta = 100$. Elements included in the patches are highlighted in darker colors. (a) Initial extent. (b) Extent at timestep 1,098,072. (c) Largest extent (occurring at timestep 1,203,758) (d) final extent (reached at timestep 1,374,294)

Traditional efficiency of ADCIRC++, which is not optimized yet, is only about 15% percent less than the efficiency of original ADCIRC. The gain from ASM, which could not be implemented in original ADCIRC (at least as efficiently), however, is overwhelming: A typical child domain run requires less than 1% of computational resources required for a conventional ADCIRC run. (All runtime comparisons are made on a 64-core AMD Opteron Processor 6274 workstation using the most recent GNU compilers for C++ and Fortran with the optimization flag \texttt{-O2}.)

In addition to extensibility, data abstraction provides ADCIRC++ with maintainability and ease of debugging. In original ADCIRC, tracking down a bug may require developers to search large parts of the source code since data structures are not designed with representation independence and data abstraction in mind, and so a local change affects the model data globally.
3.7 Conclusions

Numerical ocean models based on procedural programming and global data structures are hard to maintain and extend, since structuring the code with respect to routines operating over model data, instead of the data to operate over, increases the complexity of the program and complicates the process of adding new features or refining and optimizing the existing ones. Moreover, procedural programming languages, such as Fortran and C, lack reentrant data structures and dynamic mechanisms that can facilitate the development of next-generation approaches. Object oriented programming, on the other hand, improves the maintainability and extensibility of numerical models through data abstraction and encapsulation, and provides mechanisms to enhance dynamic behavior.

The concept of adaptive subdomain modeling, a computational technique for multi-analysis of ocean models, necessitated a new software architecture to be designed for ADCIRC ocean model, since the existing structuring of the code is based on procedural programming and global non-reentrant data structures that complicate, and even prevent, the enhancements for the application of concurrent domain instances and dynamic grid behavior.

Consequently, we designed a new software architecture based on object oriented programming and data abstraction, with dynamic and generic data structures, that provide modularity, maintainability, extensibility, and compliance with dynamic behavior. As a first prototype, we developed a simple finite-volume shallow water equations solver with the new architecture to test and refine both the computational method and the software architecture. Subsequently, we re-implemented ADCIRC with the new architecture, and incorporated the computational method in the model. Test results show that both the method and the architecture are highly efficient.

The architecture is generalizable to a variety of ocean models based on different formulations and numerical methods, with the ability to accommodate effectively the next-generation approaches including adaptive mesh refinement and adaptive subdomain modeling. Additionally,
it possesses the well-acknowledged advantages of object oriented software design.

Future directions include refinements in the threading implementation of the generic framework, such as the support for load balancing, and, for ADCIRC++, the incorporation of omitted capabilities, such as hot-starting and SWAN wave model coupling.
Chapter 4

Model Checking Wetting and Drying Algorithms

Abstract Wetting and drying algorithms of ocean models that determine the spatial extent of hydrodynamic activity bring about complex algorithmic behavior. The assessment of critical algorithmic properties by testing or deductive reasoning is prohibitive and/or prone to be misleading. As an alternative verification approach for discrete algorithmic behavior in numerical models, we propose the utilization of model checking tools and introduce an abstraction and modeling guideline for wetting and drying algorithms. We present a verification model of the wetting and drying algorithm of ADCIRC, a widely used hydrodynamic model, and test its reliability for a recent performance enhancement technique. We also provide a brief introduction to Promela modeling language and SPIN model checker which we use for our test cases.

4.1 Introduction

Along with algebraic and numerical algorithms, models of physical phenomena may incorporate auxiliary, decisional algorithms based on the assessment of physical, empirical and/or computational criteria to determine and alter the manner in which the discrete computations are carried out over a computational grid. Examples of such algorithms include wetting and drying algorithms of hydrodynamic models, which is the focus of this study, adaptive mesh refinement routines, grid adaptivity algorithm of a recently developed performance enhancement technique [3], and mesh adaptation algorithms for dynamic load balancing implementations. During run-
time, these algorithms take the grid topology and the numerical solution into account and adjust the computations, for instance, by refining/coarsening the grid at a particular location or by changing the spatial extent of the computations with moving boundaries. The verification of the logical correctness of such decisional and discrete algorithms is, therefore, highly critical for the overall reliability and robustness of a numerical model.

Due to the geographic extent of meteorological, atmospheric, and oceanographic processes, numerical models of physical phenomena operate over large parts of the globe. As a result, computational grids of such models consist of large numbers of discrete components; nodes, elements, cells, etc., usually with complex geometries. Data dependencies of discrete grid components and their interactions during the execution of such decisional algorithms, therefore, lead to overwhelming complexity in the analysis of algorithms in question, which may further be aggravated by the coexistence of multiple algorithmic model components altering the discrete computations, e.g., a wetting and drying algorithm and an adaptive mesh refinement routine executed simultaneously over a computational grid.

In practice, two common approaches for analyzing the behavior and reliability of such decisional algorithms of numerical models are testing and deductive reasoning. The former is efficient in detection of distinct issues. However, given the variety of model configurations and sizes of typical computational grids, e.g., the Western North Atlantic Mesh consisting of over 1.2 million elements (Figure 4.1), testing does not guarantee to reveal all potential issues. Deductive reasoning, on the other hand, is time-consuming and requires expertise in logical reasoning [28].

As an alternative verification approach for such algorithms of numerical models, we propose the application of model checking tools, which are commonly used to verify safety and liveness properties of distributed and concurrent systems where multiple components interact in complex and often unpredictable fashion. Designers can investigate and verify the correctness of such systems by modeling the relevant aspects, e.g., communication procedures between components, and by abstracting away from less pertinent implementation details. An effective abstraction
approach, combined with exhaustive state-space exploration strategy of model checking tools, can guarantee an efficient and reliable verification procedure for critical decisional algorithms of numerical models.

Wetting and drying schemes, typical examples of such critical algorithms, are incorporated into hydrodynamic models to determine the movement of wetting fronts in coastal regions and river floodplains in the events of rising and falling tides, storm surge inundation, tsunamis, and dam-break floods. Depending on a number of criteria (usually regarding water surface elevations and velocities) that are evaluated over the grid, these algorithms designate the discrete grid components to be wet, partially wet, or dry, and so determine the spatial extent of hydrodynamic activity. A non-trivial challenge in the development of wetting and drying algorithms from the software engineering point of view is the assessment of algorithmic behavior that is determined by the interdependencies between adjacent grid components and complex control flows embedded into the code to represent the physical processes [31].

To verify the correctness of wetting and drying schemes and to determine whether they conform and ensure the asserted algorithmic properties, we introduce an abstraction and model
checking guideline. We present a case study where we construct a verification model of the wetting and drying algorithm of ADCIRC, a widely used advanced hydrodynamic model, using SPIN model checker. We use the verification model to confirm the existence an undesired behavior in the algorithm, initially discovered by ADCIRC developers through test runs. We then verify the correctness of wetting and drying for a recently developed exact reanalysis technique in ADCIRC, called subdomain modeling.

The remainder of this paper is organized as follows. In Section 4.2, we discuss the motivation behind the approach and present several examples of potential applications. In Section 4.3, we provide a brief background information on ADCIRC and the subdomain modeling approach, and an introduction to SPIN and Promela. In Section 4.4, we outline an abstraction and modeling procedure for discretized numerical models and wetting and drying routines. In Section 4.5, we present the SPIN verification model of the wetting and drying algorithm of ADCIRC, and verify a couple of properties including the correctness of wetting and drying for subdomains. In section 4.6, we discuss the limitations of the guideline. In Section 4.7, we provide a brief overview of the previous studies on validation of wetting and drying algorithms and verification of numerical models. Finally, we discuss the conclusions.

4.2 Motivation

The performance and accuracy of wetting and drying schemes are of great importance for the overall reliability of hydrodynamic models. While there is a considerable effort towards the validation of physical and numerical properties of wetting and drying algorithms, verification methods and approaches for the decisional and logical properties regarding software and algorithm development aspect of wetting and drying schemes, and other similar routines, have not been addressed in literature. In this section, we briefly define the physical and numerical concerns of the developers of wetting and drying algorithms, and list several approaches to address these concerns. We then discuss the current and potential issues regarding decisional and discrete algorithmic behavior, which can be examined with model checking tools, as shown
in the following sections.

Generally, physical challenges encountered by the developers of wetting and drying algorithms involve accurate representation of hydrodynamic processes defining shoreline evolution over complex topography and bottom surface conditions [10]. The discrete and nonlinear nature of wetting and drying schemes complicate the physical modeling of mass and momentum conservation along wetting fronts, so modelers use analytical and numerical approaches to validate their schemes. Horritt [58], for instance, uses numerical solutions of simplified flow equations to assess the performance and accuracy of several wetting and drying algorithms for TELEMAC-2D, a two-dimensional model for shallow water simulations [46], with respect to their representation of water surface and mass conservation properties. In another study, Marche et al. [77], validate the wetting and drying procedure of their nonlinear shallow water model by comparing the model results with analytical solutions to assess its performance, while Oey [82] extends a previously implemented wetting and drying scheme in POM, an ocean circulation model [20], to three-dimensional simulations and tests the scheme for its conservation properties. In addition to physical challenges, numerical challenges posed by the wetting and drying algorithms involve stability, spurious oscillations, numerical diffusion, etc. Bradford and Sanders [23], for instance, develop a finite-volume shallow water model based on Roe approximate Riemann solver, and discuss methods to resolve numerical instability and diffusion associated with partially wet cells. A comprehensive review of studies on wetting and drying algorithms is presented by Medeiros et al. [79].

With respect to decisional and discrete algorithmic properties of wetting and drying schemes, an example reliability concern can be observed in the ADCIRC implementation, where the wet/dry state of a node depends not only on its own properties but also the properties of adjacent nodes and elements. In the parallel version of ADCIRC, which employs a domain decomposition strategy that requires the exchange of properties between neighboring subdomains through ghost nodes [90], the question of how frequently and at which control points these states must be exchanged while the algorithm is being executed requires careful consideration
to ensure that the algorithm can faithfully determine the wet/dry states of nodes adjacent to ghost nodes.

A mass-conserved wetting and drying algorithm implemented in POM introduces a similar complexity. As described by Xie et al. [96], a set of wetting and drying criteria are checked to determine the states of grid cells. According to the algorithm, cells which are not adjacent to a wet cell are prevented from becoming inundated. In a potential nesting implementation or a domain decomposition for parallelism, such data dependencies of wetting and drying procedure are not maintained in the cells along the boundary, and so require additional measures to accommodate the lost adjacency, complicating the analysis of the algorithm.

FVCOM, an unstructured-grid finite volume ocean model, employs a relatively simple wetting and drying algorithm where a node is determined to be wet if its water height is greater than a specified minimum water height, and an element is determined to be wet if the sum of minimum of the bathymetric heights and maximum of the surface elevations of nodes incident on the element is greater than the specified minimum water height [27]. Although wet/dry states of nodes depend only on their own water heights and wet/dry states of elements depend only on bathymetric depths and water surface elevations of nodes incident on them, the assessment of the logical and temporal properties may become non-trivial in case an additional decisional algorithm is incorporated. An adaptive mesh refinement routine, for instance, would require careful handling of interactions between nodes and elements of coarse and refined grids for a robust wetting and drying algorithm.

4.3 Background

4.3.1 ADCIRC Background

ADCIRC is a finite-element hydrodynamic model used to simulate the effects of tides and hurricane storm surge on civil infrastructure. The governing equations and optimized numerical algorithms of ADCIRC provide efficiency and high accuracy while unstructured triangular grids
provide enough flexibility to represent coastal features with complex geometries [74].

ADCIRC simulations can be performed in three-dimensional or in two-dimensional depth-integrated (2DDI) configurations. The spatial discretization is based on a continuous Galerkin formulation, and the time discretization may be configured to be implicit or explicit. The coefficient matrix of the linear system can be assembled as either a consistent or a lumped mass matrix. In case of consistent matrix assembly, the system is solved using an iterative Jacobi Conjugate Gradient solver. In our studies, we use the 2DDI version of ADCIRC with consistent matrix solver and implicit time discretization.

At every timestep, ADCIRC solves the Generalized Wave Continuity Equation (GWCE) to obtain the water surface elevations at grid nodes. Following the GWCE, ADCIRC evaluates the wetting and drying algorithm to determine the nodes and elements participating in computations depending on several criteria. Finally, ADCIRC solves the momentum equations to obtain the velocities.

**Wetting and drying algorithm**

The wetting and drying algorithm in ADCIRC consists of four main criteria, depending mainly on the water surface elevations, that are sequentially evaluated for all the nodes and elements of a grid to determine their wet/dry states. Before describing the algorithm in detail, we begin with the following definitions [12].

**Final wet/dry status** the wet/dry status of a node that is obtained after the evaluation of the wetting and drying algorithm is completed. A final wet/dry status remains the same until it is reevaluated at the next timestep.

**Temporary wet/dry status** the wet/dry status of a node which is subject to frequent changes during the evaluation of the algorithm.

**Intermediate wet/dry status** Temporary wet/dry status of a node between the second and the third steps of the algorithm. Intermediate wet/dry state may or may not be the same
as the final wet/dry state, depending on the potential changes in the remaining steps.

**Active element** A wet element is said to be active if it has three wet nodes. A dry element or a wet element with at least one dry node is labeled as inactive.

At every execution of the algorithm, all elements are initially set to be wet. In the first step of the algorithm, wet nodes with water columns heights less than the minimum water column height $H_0$ are made dry. Then, in the second step, the steady state velocity that would result from a balance between the water level gradient and bottom friction is calculated at each element with exactly two wet nodes. If the steady state velocity is greater than the minimum wetting velocity at an element with two wet nodes, the remaining dry node is made temporarily wet. Note that, since the steady state velocity mostly depends on the water surface elevations, this criterion is rather a height restriction [34]. Once the nodal wetting criterion is evaluated, each element is checked in the third step to determine whether any of the two nodes with higher water surface elevations is barely wet, i.e., has a water column height less than $1.2H_0$, and if so, the element is made dry. In the final step, wet nodes that are connected only to inactive elements are considered as landlocked and are made dry. After all of the criteria are evaluated, the final wet/dry states are updated. Algorithm 1 shows the summary of the wetting and drying scheme in ADCIRC.

### 4.3.2 Subdomain Modeling

Subdomain modeling approach is an exact reanalysis technique to reduce the computational effort for repeated simulations of numerical ocean models, where each recurrence includes a local change corresponding to an engineering design or failure scenario [12]. The technique is based on applying the local changes to subdomain grids extracted from a sizable original grid, called full-scale grid, and enforcing the boundaries of subdomains using the data obtained from the outputs of the full-scale grid. Boundary conditions for subdomains include the water surface elevations, wet/dry states, and velocities. The technique is incorporated in the official ADCIRC
Algorithm 1 Wetting and Drying

0: for e in elements do
    make e wet
    \(\triangledown\) start with all elements being wet

1: for n in nodes do
   if \(W_n\) and \(H_n < H_0\) then
      \(W_n \leftarrow \) false, \(W^t_n \leftarrow \) false
   \(\triangledown\) make nodes with low water column height dry

2: for e in elements do
   if e has exactly 2 wet nodes and \(V_{ss}(e) > V_{min}\) then
      let \(j\) be the remaining dry node
      \(W^t_j \leftarrow \) true
     \(\triangledown\) propagate wetting unless slow flow

3: for e in elements do
   find nodes \(i\) and \(j\) of \(e\) with highest water surface
   if \(H_i, H_j < 1.2H_0\) then
      make element \(e\) dry
   \(\triangledown\) allow water to build up on downhill slopes

4: for n in nodes do
   if \(W^t_n\) and \(n\) on only inactive elements then
      \(W^t_n \leftarrow \) false
   \(\triangledown\) make landlocked nodes dry

5: for n in nodes do
   \(W_n \leftarrow W^t_n\)
   \(\triangledown\) set the final wet-dry state for nodes

release and is used by agencies and research groups for science and engineering applications. A user interface, called SMT, is developed to automate the workflow of the technique [42]. An example subdomain extraction using SMT is shown in Figure 4.2.

The accurate prescription of the boundary conditions is crucial for the reliability of the technique, especially for the wet/dry states since the execution of the wetting and drying algorithm of ADCIRC is influenced by adjacencies of nodes and elements (Figure 4.3). Since the adjacencies around the boundaries change after a subdomain extraction, a careful consideration is necessary to determine the control points at which the wet/dry states of subdomain boundary nodes are enforced during the execution of the algorithm. Using deductive reasoning, we previously proved that enforcing both the intermediate wet/dry states after the second step and final wet/dry states after the fourth step using the final wet/dry states obtained from the full domain is sufficient to reconstruct the correct wet/dry states at and near the subdomain boundaries [12]. We reaffirm the correctness of the wetting and drying for subdomains in Section
4.5.4, by developing an abstract representation of the algorithm for the SPIN model checker.

An updated subdomain modeling approach is recently developed, which is realized by the concurrent execution of an original full scale domain, called parent domain, and multiple child domains with local modifications [3]. A fundamental difference between the conventional subdomain modeling (CSM) approach and the new approach, called *adaptive subdomain modeling* (ASM), is that the subdomain boundary locations in CSM are static and determined by the user prior to the simulation, whereas, in ASM, the extent of child domains are determined automatically and adjusted adaptively during runtime to ensure that the computations are carried out only at the regions with altered hydrodynamics originating from the local changes.

The verification model presented in Section 4.5.4 applies to both methods for three reasons. First, the boundary of a child domain in ASM is moved before a timestep begins and stays the same until the next timestep. Second, the wetting and drying algorithm depends only on the properties determined at the timestep in which the algorithm is evaluated. Finally, the two methods enforce the boundary conditions at the same exact control points.
4.3.3 SPIN and Promela

SPIN model checker is a formal verification tool that is widely used to verify hardware and software systems with concurrent and interacting components [55]. SPIN takes a verification model, which is an abstract representation of a system, as input and analyzes it in simulation mode or in verification mode. While simulation mode is helpful for detecting syntax errors and observing the general behavior, verification mode ensures that all possible execution sequences are exhaustively searched to check if the properties in question hold.

SPIN inputs are written in Promela, a C-like verification modeling language. Although resembling imperative programming languages, Promela only includes basic data types, except floating-point types, and essential control flow statements along with functional object types such as message channels. The minimalistic development environment of Promela directs mod-
ler to develop an abstract representation of the actual system and helps SPIN analyze the model more efficiently. In the remainder of this section, we provide an introductory overview of the features and mechanisms in Promela that we use for our wetting and drying model. For complete descriptions of SPIN and Promela, we refer readers to Holzmann [56] and Ben-Ari [15].

In Promela, the behavior of a system is modeled by the statements and expressions declared within process blocks. SPIN processes are similar to functions in programming languages, though, one fundamental difference is that multiple active processes in a SPIN model are executed asynchronously, i.e., the order in which the statements of concurrent processes is executed is non-deterministic.

Figure 4.4 demonstrates an introductory verification model with three key components: a global variable to be modified, a SPIN process, and a safety property to be checked. Following the declaration of the global integer variable $x$ with an initial value of zero, the process named increment is declared with the keyword active, which signifies that the process is to be automatically initialized during runtime. Within the process, a local variable $x_{\text{loc}}$ is initialized to the value of the global variable $x$. Subsequently, $x$ is set to the value of the local variable plus one, in effect, incrementing the original value of $x$ by one. The safety property, which is expressed as a linear temporal logic (ltl) formula, is self-explanatory: for all possible execution sequences, the final value of $x$ must always be equal to one. Since this trivial example does not include non-determinism, which would either result from process interleavings or from non-deterministic branches, there is only one execution sequence (resulting in $x$ being equal to one) and no property violation.

We now update the process declaration and the safety property as shown in Figure 4.5 to model two concurrent processes incrementing the value of the global variable. (The number between the square brackets in the process declaration instructs SPIN to generate given amount of concurrent instances of the same process). This time, the verification analysis of the model reveals an execution sequence violating the safety property, i.e., where the final state is $x \neq 2$. The violation is due to the fact that process $i$, one of the two concurrent processes, may be
// A global variable:
int x = 0;

// A SPIN process:
active proctype increment(){
    int x_loc = x; // copy the value to a local variable
    x = x_loc+1;   // update the global variable
}

// A linear temporal logic property:
ltl {always eventually (x==1)}

Figure 4.4: A global variable, a SPIN process, and a safety property.

unaware of the incrementation applied by the other process j if the incrementation by the process j is applied after the initial value of x, which is zero, is copied to the local variable of i. As a result, both processes set the global variable to one.

// Concurrent SPIN processes:
active [2] proctype increment(){
    int x_loc = x; // copy the value to a local variable
    x = x_loc+1;   // update the global variable
}

// A linear temporal logic property:
ltl {always eventually (x==2)}

Figure 4.5: Concurrent incrementation.

A straightforward solution to fix this issue is to make the incrementation process atomic, i.e., uninterruptable. To demonstrate the executability rule in SPIN, however, we implement a different solution. First, we declare an additional global variable, turn, of type integer, initially set to zero. This additional variable is to determine the order in which the concurrent processes are executed. Then, we modify the definition of the process increment, as shown in Figure 4.6. The first statement of the updated process compares turn and the ID of the process instance, which is retrieved using the keyword .pid. In SPIN, the ID of each process instance is sequentially assigned starting from zero. Therefore, initially, the first statement evaluates to
true for only the process whose ID is equal to zero. By executability rule in SPIN, a statement blocks a process until it evaluates to true. Consequently, it is guaranteed that the process whose ID is equal to one cannot execute its remaining statements until the other process increments turn by one at its final statement. This basic lock mechanism, implemented using the shared variable turn, prevents a process from modifying the other shared variable x without being noticed by the other process. Notice that turn is incremented atomically, and, therefore, poses no race condition.

```
 active [2] proctype increment(){
  turn == _pid;  // check if it is current processor’s turn
  int x_loc = x;  // copy the value to a local variable
  x = x_loc+1;    // update the global variable
  turn++;         // let the next processor execute its statements
}
```

Figure 4.6: Synchronized incrementation.

The executability rule is also used to create selection structures where one or more guard statements can be listed to introduce branches at a control point. A selection structure begins and ends with the keywords if and fi, respectively, and a list of guards are provided in between, where each guard is preceded by a double colon. Statements that are to follow a guard statement are placed after the statement separator (->), which is equivalent of a semicolon. In a selection structure, if only one of the guards evaluate to true, the process continues with executing the statements following that guard. If multiple guards evaluate to true, a non-deterministic choice is made: In simulation mode, SPIN randomly chooses one of the guards evaluating to true, whereas in verification mode, all possible execution sequences are evaluated to check whether any counter-example exists. In Figure 4.7, we rewrite the error-free, concurrent process increment using a selective structure with a single guard (turn == _pid).

Repetition structures, which begin and end with the keywords do and od, similarly contain guard statements to determine which branch to follow. A repetition statement is executed
active [2] proctype increment()
{
    if :: turn == _pid ->
        int x_loc = x; // copy the value to a local variable
        x = x_loc + 1; // update the global variable
    fi
    turn++; // let the next processor execute its statements
}

Figure 4.7: Synchronized incrementation with a selective structure.

repeatedly until the control passes to the keyword break. The keyword else, as a guard statement, evaluates to true if and only if all the remaining guards evaluate to false. In Figure 4.8, we remodel the error-free, concurrent process increment using a repetitive structure. Note that for-loops may alternatively be used for repetitions with predetermined number of steps as we frequently do in our wetting and drying model.

active [2] proctype increment()
{
    do :: turn == _pid ->
        int x_loc = x; // copy the value to a local variable
        x = x_loc + 1; // update the global variable
        break; // end loop
    :: else -> skip; // busy-wait
    od
    turn++; // let the next processor execute its statements
}

Figure 4.8: A busy-wait implementation using a repetitive statement.

The syntaxes of selection and repetition structures in Promela are similar to that of the Guarded Command Language introduced by Dijkstra to define alternative and repetitive constructs that allow non-deterministic choices [39]. However, a difference is that in Dijkstra’s language, if all guards are false, an alternative (selection) construct aborts and a repetitive construct terminates [39], whereas, in SPIN, the process blocks until one of the guards evaluate to true [56].
Instead of shared variables, e.g., turn in the previous examples, concurrent processes may be synchronized using channels that allow processes to exchange messages. Channels in Promela are declared with a buffer size (zero or more) and message fields (one or more) that designate the types of variables to be sent. A process sends and receives a message using the send operator (!) and receive operator (?). Both operators are preceded by the name of the channel and followed by the message to be sent or received. If the buffer of a channel is full, a receive statement blocks the process until the buffer has space. If the buffer size is declared to be zero, send and receive operations are performed synchronously.

In Figure 4.9, we declare a message channel, named lock with a buffer size of one message of type bool, and modify the process increment to use the channel for synchronization. Rather than exchanging a message between processes, we use the channel as a lock to ensure that only one of the concurrent process enters its critical section where it reads and writes to the global variable. Since the message channel can only store one message, only one of the processes can send a message through lock, and the other process waits until the active process empties the buffer by receiving its own message. (For this example, the value, true, that is sent and received through the channel has no effect in the behavior of the processes.) Notice that, contrary to the previous examples, the order in which the processes increment the global variable is nondeterministic and unaffected by process IDs.

```
// Declare a message channel
chan lock = [1] of {bool};

active [2] proctype increment(){
    lock ! true; // send "true" thru channel "lock"
    int x_loc = x; // copy the value to a local variable
    x = x_loc+1; // update the global variable
    lock ? _; // receive any msg thru channel "lock"
}
```

Figure 4.9: A message channel used as a lock.
Several features of message channels further increase their practicality. A message may be copied without removing it from the channel buffer by enclosing the list of variables between angle brackets in a receive statement [15]. In a similar fashion, square brackets may be used to check whether a receive statement is executable, as done in Figure 4.25. By default, a message channel behaves as a queue, i.e., a FIFO data structure. However, using the sorted send operator (!!), or random receive operator (?), the default behavior may be modified. Note that the random receive operator returns the first message in a buffer that satisfies the executability rule [56]. An example usage of random receive operator may be seen in Figure 4.25. If a constant value, either hard-coded or evaluated using the function eval(), is provided for one of the fields in a receive statement, message passing can only occur if the constant value matches the value of the message. This behavior can be used to classify the messages in a channel buffer or to determine the address of the recipient.

Other functionalities of Promela that we use for our wetting and drying model include macros (Figure 4.11), inline definitions which behave similar to pass-by-reference functions of programming languages, and compound data structures used to introduce user-defined data types that are the compositions of basic types, such as the ones defined in Figure 4.13. We also use atomic sequences declared with the keyword atomic, and deterministic sequences declared with the keyword d_step which encapsulate multiple statements and execute them as though they are a single statement. While the main function of both sequences is to prevent interleavings, we use them to improve the efficiency of the verifier. Note that a restriction in deterministic sequences is that they cannot include non-deterministic choices.

Though achievable for simple examples like the ones presented thus far, developing a verification model as a mere copy of instructions of the actual system is infeasible and overwhelming. Therefore, in practice, a verification model is developed as an abstract representation of the actual implementation, as done in the following sections.
4.4 Abstracting Wetting and Drying Algorithms

For a decisional algorithm of a numerical model, the two aspects to be captured in a verification model are (1) the discrete time and space the algorithm operates over, and (2) the data and control flow of the algorithm. Since model checkers can only verify systems consisting of a finite number of states and transitions, both aspects must effectively be abstracted to establish a tractable model while ensuring reliability. In this section, we introduce a general abstraction guideline for these two aspects of wetting and drying algorithms.

4.4.1 Discretization in Space and Time

One of the main sources of complexity in decisional algorithms of numerical models is that such algorithms operate over a discretized space, which may consist of millions of nodes and countless variations of grid topologies. To avoid under-approximation, i.e., lack of a behavior existing in the actual system, the abstract grid in the verification model must be representative of all possible mesh configurations, and, for efficiency, it should include as few discrete components (nodes, cells, elements, etc.) as possible.

In this study, the approach that we propose for representing the discretization in space is based on constructively defining an abstract grid that includes the least number of internal components sufficient to express the properties to be checked, and additional interface components surrounding the internal ones. While the internal components of the abstract grid allow the modeler to incorporate enough detail about the system behavior, the interface components close the system by encircling the abstract grid. For the abstract grid to represent a subset of any possible grid, the following three conditions must hold.

If the first condition is satisfied, it is guaranteed that the external components that would exist in an actual grid but are left out of abstract grid cannot affect the behavior of internal components within a timestep. The second condition is to guarantee that the behavior of interface components is not under- or over-approximated. The final condition complements the first
1. The abstract grid encompasses the domain of dependence of all internal components.

2. The behavior of interface components are adjusted to account for the effects of external components left out of the abstract grid.

3. The set of number of interface components in abstract grid include all possible numbers of corresponding components that can occur in reality.

Figure 4.10: Conditions for an abstract grid to represent a subset of any possible grid.

two conditions to ensure that the abstract grid represents a subset of any realizable grid.

Of all three conditions, perhaps the most challenging one to satisfy is the second condition. Since the abstract grid does not include external components that would be adjacent to interface components, the algorithmic behavior of interface components must be adjusted at the control points of the algorithm where the state of external components would affect the state of interface components. This may be done in the abstract model by introducing non-deterministic choices at those control points to provide all possible branches for interface components. Closing the model in this manner is similar to compositional reasoning, and particularly assume-guarantee reasoning [28], though two properties of wetting and drying algorithms simplify our environment modeling problem. First, the components left out of the verification model are of the same type and subject to the same algorithm as the ones within the model. Second, wetting and drying algorithms are evaluated sequentially for each discrete component, and therefore, there is no need to consider interleavings and random interactions between components. Nevertheless, non-deterministic choices to account for the effects of components left out of the verification model must be carefully implemented to avoid under- or over-approximation.

With respect to time discretization, we propose an abstraction approach that is based on initializing the states of discrete components non-deterministically, such that the set of these choices corresponding to initial states within a timestep is also the superset of all possible states that can be reached after an execution of a timestep. This approach ensures that the analysis
of the verification model covers all possible consecutive executions of infinitely many timesteps.

4.4.2 Data and Control Flow

Once the numerical discretization is abstracted, the wetting and drying algorithm and additional routines affecting the behavior can be incorporated in the verification model using various abstraction techniques at present. The control flow of the algorithm can be expressed using compound statements and other mechanisms available in the preferred modeling language, while discrete data describing and influencing the algorithm, such as water surface elevations, flux velocities, and wet/dry states, may be incorporated into data structures representing the discrete mesh components. For properties with a large range of values, data-type abstraction may be used to reduce the value range [56]. Similarly, predicate abstraction, which is based on abstract interpretation, can be employed to represent the states of components’ properties using various boolean predicates [92].

An example application of such abstractions can be seen in the second part of our verification model of ADCIRC’s wetting and drying algorithm: In case the internal node is dry and is connected to at least one element whose remaining nodes are wet, the verification model non-deterministically chooses whether the internal node should also be made wet or not. In the original wetting and drying algorithm, this decision is made deterministically by first calculating the steady state velocity resulting from a balance between the water level gradient and bottom friction, and then comparing it with a minimum velocity. Since our aim is to detect an undesired behavior in the algorithm that would result from engaging in either choice, we only introduce a non-deterministic choice of velocity being greater than the minimum velocity, or not, and abstract away from the details of how the steady state velocity is calculated and compared.

Once a complete verification model of the discretization, data, and control flow of the algorithm is developed for a property to be checked, various additional properties may be examined sequentially by making only minor adjustments to the original verification model: Depending on the new property to be checked, additional details may be incorporated into the
model, or details insignificant for the new property may be left out to reduce the size of the state space. In summary, the abstraction approach we propose for verifying wetting and drying algorithms of numerical ocean models is as follows:

1. Generate an abstract representation of the grid.
   1.1. Incorporate as few internal components as possible but sufficient to express the properties to be checked.
   1.2. Generalize the abstract grid by incorporating interface components according to the conditions listed in Figure 4.10.

2. Generate an abstract representation of time discretization.
   2.1. Ensure that the set of initial states is the superset of any potential final state of a timestep execution.

3. Model the data and control flow of the decisional algorithm.
   3.1. Abstract away from insignificant implementation-specific details of the algorithm using methods in practice, such as predicate abstraction.
   3.2. Adjust the abstraction of the data and control flow for additional properties to be checked.

4.5 Case Study: ADCIRC’s wetting and drying algorithm

Following the abstraction guideline presented in Section 4.4, we develop a verification model of ADCIRC’s wetting and drying algorithm to examine two properties. First, we determine in what circumstances an element with three wet nodes can become dry, a question previously answered by ADCIRC developers through test runs. The second property we examine is the reliability of the subdomain modeling approach. Specifically, we would like to verify that the nodes adjacent to subdomain boundaries have the correct wet/dry states for all configurations.
4.5.1 Abstract Grid

As discussed in Section 4.4, the number of internal and interface nodes of an abstract grid depends on the properties to be checked and the variety of potential grid topologies. Since both properties to be checked can be expressed on an abstract grid with as few as a single internal node, we only include one internal node encircled by a number of interface nodes. For the abstract grid to represent a subset of any unstructured triangular ADCIRC grid, we determine a minimum and a maximum number of interface nodes. By geometry, the minimum number of nodes that can be placed around the internal node is three. Although there is not a limit on the maximum number of adjacent nodes, we choose to set the maximum number of nodes, including the internal one, to eight, which is consistent with meshes used in practice, since as the number of adjacent nodes increases, the mesh quality and, therefore, the numerical stability degrades. Since the number of nodes is non-deterministically set to be any value from \( \text{min} \) to \( \text{max} \) in the macro \texttt{setSizes()}\), shown in Figure 4.11, SPIN exhaustively constructs abstract grids for all cases to check whether a counter-example exists. In Figure 4.12, abstract grids with the minimum and the maximum number of nodes are shown.

```
// Macros of min and max number of nodes and elements:
#define nn_min 4       // min number of nodes
#define nn_max 8       // max number of nodes
#define ne_min (nn_min-1) // min number of elements
#define ne_max (nn_max-1) // max number of elements

// Global variables:
byte nn;  // number of nodes
byte ne;  // number of elements

// Non-deterministically sets the number of nodes:
#define setSizes() \n    select(nn:nn_min..nn_max); ne=nn-1; \n    printf(" Number of nodes: %d\n",nn);
```

Figure 4.11: Determining the size of the abstract grid.
We model the nodes and elements as compound data structures. A node in our verification model consists of three boolean properties, \texttt{wet}, \texttt{tempWet}, and \texttt{mju}, to designate whether the node is wet, temporarily wet, and adjacent to an active element, respectively. Similarly, an element consists of a boolean property \texttt{wet}. The arrays of nodes and elements are encapsulated by another compound data type, named \texttt{Domain}. Figure 4.13 show the definitions of these compound types.

In ADCIRC, node indices of each element are stored in adjacency matrices. In order to reduce the number of variables, and so to increase tractability of the verification model, we adopt a fixed indexing approach for the nodes and elements of the abstract grids. The index of the internal node is always zero, and the indices of interface nodes sequentially increase as they are placed around the internal node. For an element with index i, the index of the first incident node is always zero, which corresponds to the internal node, the index of the second incident node is i+1, and the index of the third incident node is obtained by the formula \((i+2)\%nn+(i+2)/nn\). This indexing approach for nodes and elements can be observed in the abstract grids shown in Figure 4.12. The indices of incident elements for element \texttt{i} is retrieved.
typedef Node{
    bool tempWet;
    bool wet;
    bool mju=false;  // true if node is adjacent to an active element
}

typedef Element{
    bool wet=true;
}

typedef Domain{
    Node nodes[nn_max];
    Element elements[ne_max];
}

Figure 4.13: Compound data types: Node, Element, Domain.

using the macros shown in Figure 4.14.

#define n1(i) 0
#define n2(i) i+1
#define n3(i) (i+2)%nn+(i+2)/nn

Figure 4.14: Macros to retrieve incident nodes of element i.

4.5.2 Control Flow

As described in Section 4.3.1, ADCIRC’s wetting and drying algorithm consists of four main criteria evaluated sequentially to determine wet/dry states of nodes and elements. Before modeling the control flow of each criterion as separate inline functions, we begin with timestep initialization routine. Since the elements in ADCIRC are set to be wet at the beginning of every timestep, wet/dry states of elements in the verification model are also initially set to be wet. To ensure that the initial wet/dry states of nodes are the superset of states that would result from any execution of the previous timestep, the wet/dry states of nodes are non-deterministically initialized. Figure 4.15 shows the inline function initTimestep() including a for-loop to iterate over all nodes. Since both guards of the selection structure, makeDry(i) and makeWet(i),
always evaluate to true, the choice for each node is non-deterministic. Note that the definitions of auxiliary inlines and macros are provided in Appendix B.

```c
inline initTimestep(domain) {
    printf("Initializing nodes\n");
    for (i:0..nn-1) {
        if :: makeWet(domain, i) :: makeDry(domain, i)
            fi
    }
}
```

Figure 4.15: Timestep initialization routine.

**Part 1: Nodal Drying**  In the first stage of the algorithm, wet nodes with water heights less than the minimum water column height are made dry. In the verification model, this behavior is abstracted by providing a non-deterministic choice for wet nodes: drying or remaining wet. During verification, SPIN exhaustively checks for both choices to detect whether any violation occurs when engaging in either choice. The inline function `part1()` is shown in Figure 4.16.

```c
inline part1(domain) {
    printf("Part-1: Nodal Drying\n");
    for (i:0..nn-1) {
        if
            :: isWet(domain, i) ->
                if
                    :: skip; // remain dry
                    :: makeDry(domain, i);
                fi
            :: else -> skip
        fi
    }
}
```

Figure 4.16: SPIN model of the first stage of the wetting and drying algorithm.
**Part 2: Nodal Wetting**  In the original implementation of the second part of the algorithm, the steady state velocity resulting from a balance between the water level gradient and bottom friction is calculated in each element with exactly two wet nodes. If the velocity is greater than the minimum velocity, the remaining dry node is made wet. Our abstract control flow for this stage of the algorithm (Figure 4.17) provides a non-deterministic choice for a dry internal node to become wet or remain dry if it is connected to at least one element with two wet (interface) nodes. Since this criterion depends on the properties of adjacent elements of a node, the modeled behavior of interface nodes need to account for the potential effects of external nodes that are left out: Dry interface nodes can be made wet even if they are not connected to an element with two wet nodes since external elements may cause them to be made wet in reality.

```c
inline part2(domain){
  printf("Part-2: Nodal Wetting\n");
  bool excludeInternalNode = true;
  if :: isWet(domain,0) -> skip
  :: else ->
    for (i:0..ne-1){
      if :: ( isWet(domain,n2(i)) && isWet(domain,n3(i)) ) ->
        excludeInternalNode = false; goto evalPart2
      :: else -> skip;
      fi
    }
  fi

evalPart2:
  for (i:excludeInternalNode..nn-1){
    if :: isWet(domain,i) -> skip
    :: else if :: skip
      :: makeTempWet(domain,i);
    fi
    fi
  }
}
```

Figure 4.17: SPIN model of the second stage of the wetting and drying algorithm.
Part 3: Elemental Drying  In the third part, an element is made dry in case one of its two nodes that have the largest values of water surface elevations has less water height than $1.2H_0$, i.e., it is barely wet. Clearly, if an element has less than two wet nodes, the element should deterministically be made dry since one of the two nodes that have the largest values of water surface elevations is dry, and, therefore has a water column height less than $1.2H_0$. Otherwise, the element may stay wet depending on the water column heights of its two nodes that have the highest water surface elevations, a behavior modeled non-deterministically in the verification model (Figure 4.18).

```inline part3(domain){
    printf("Part-3: Elemental Drying\n");
    for (i:0..ne-1){
        if :: nWetNodes(domain,i)<2 -> makeEleDry(domain,i);
        :: else if :: skip; // let element remain wet
        :: printf(" element %d has barely wet nodes\n",i) ->
            makeEleDry(domain,i);
        fi
        fi
    }
}
```

Figure 4.18: SPIN model of the third stage of the wetting and drying algorithm.

Part 4: Landlocking  In the final part of the algorithm, wet nodes which are connected only to inactive elements are determined to be landlocked, and so are made dry. The abstract control flow of this stage begins with an elemental loop to determine the nodes that are connected to at least one active element. Similar to the second stage of the algorithm, we provide slightly different control flows for internal and interface nodes to account for the unresolved adjacencies of interface nodes. If it is determined that an internal node is wet and is connected to at least one active element, it remains wet. Otherwise, it is made dry. If an interface node is wet and connected to at least one active element, it remains wet. Otherwise, the choice of
landlocking is non-deterministic, since an adjacent element left out of the verification model may actually prevent it from being landlocked in reality. The inline function modeling the landlocking procedure is shown in Figure 4.19.

```c
inline part4(domain){
    printf("Part-4: Nodal Drying\n");

    // determine if nodes are connected to any active elements
    for (i:0..ne-1){
        if :: isElementWet(domain,i) ->
            if :: nTempWetNodes(domain,i) == 3 ->
                domain.nodes[n1(i)].mju = true;
                domain.nodes[n2(i)].mju = true;
                domain.nodes[n3(i)].mju = true;
            else -> skip;
        :: else -> skip;
        fi
    }

    // Landlocking routine for the internal node
    if :: (isTempWet(domain,0) && !(domain.nodes[0].mju) ) ->
        printf("nodes[0] is landlocked\n") -> makeTempDry(domain,0);
    :: else -> skip;
    fi

    // Landlocking routine for the interface nodes:
    for (i:1..nn-1){
        if :: (isTempWet(domain,i) && !(domain.nodes[i].mju) ) ->
            if :: printf("nodes[%d] is landlocked\n",i) -> makeTempDry(domain,i);
            :: skip; // let the interface node remain wet
        :: else -> skip; // interface node is not landlocked
        fi
    }
}
```

Figure 4.19: SPIN model of the fourth stage of the wetting and drying algorithm.
**Update** Once all four criteria are evaluated, permanent wet/dry states of all nodes are set to be the same as their temporary wet states, as shown in Figure 4.20.

```plaintext
inline update(domain){
    for (i:0..nn-1){
        domain.nodes[i].wet = domain.nodes[i].tempWet
    }
}
```

Figure 4.20: Updating permanent wet/dry states of nodes.

### 4.5.3 Elemental drying with three wet nodes

It was recently discovered by ADCIRC modelers that third stage of the wetting and drying routine, which was added to fix the mass balance problem occurring in regions with an incline where water flows from above [34], leads to erroneous station outputs and numerical issues. Although all three incident nodes of an element were wet, it was noticed that a recording station incident on the same element was recorded dry, which indicated that the element was also dry. It was then questioned whether an element with three wet nodes could become dry. Consequently, using the Shinnecock Inlet test case, available from the ADCIRC development group, it was revealed that this behavior could occur in case the element had barely wet nodes. The discovery of this behavior led the developers to add the option of deactivating the entire elemental drying criteria, which also improved the numerical stability.

To detect this behavior using the SPIN model of the algorithm, we add an assertion to the verification model, which states that elements with three wet nodes must be wet after the algorithm is evaluated (Figure 4.21). Since the verification model exhaustively checks for an execution sequence that leads to a counter-example, the assertion allows us to determine in what circumstances an element with three wet nodes can become dry.

To complete our verification model, we declare an initial process with the keyword **init**,
inline testElementalDrying(domain){
    printf("Analysis:\n");
    for (i:0..ne-1){
        if :: nWetNodes(domain,i)==3 ->
            printf(" Element[%d].wet = %d\n",i,isElementWet(domain,i));
            assert(domain.elements[i].wet)
        :: else
        fi
    }
}

Figure 4.21: An inline asserting that elements with three wet nodes must be wet.

and call all of the macros and inlines corresponding to the stages of the abstract wetting and
drying algorithm and the safety assertion.

init{
    atomic{
        // Initializations:
        setSize();
        Domain domain;
        initTimestep(domain);

        // The algorithm:
        part1(domain);
        part2(domain);
        part3(domain);
        part4(domain);
        update(domain);

        // Safety Property:
        testElementalDrying(domain);
    }
}

Figure 4.22: The initial process of elemental drying verification.

We execute the SPIN verifier. The output of the verifier reports an assertion violation. The
guided simulation of the execution trail reveals that an element with three wet nodes may be
dry if the element has a barely wet node (Figure 4.23), which coincides with the result of the
test runs previously carried out the examine the behavior.
Number of nodes: 4
Initializing nodes
Part -1: Nodal Drying
Part -2: Nodal Wetting
Part -3: Elemental Drying
  element 2 has barely wet nodes
Part -4: Nodal Drying
Analysis:
  Element[0].wet = 1
  Element[1].wet = 1
  Element[2].wet = 0
spin: wetdry.pml:202, Error: assertion violated

Figure 4.23: The guided simulation of the execution trail that results in the assertion violation.

4.5.4 Verification of wetting and drying in subdomains

The accuracy of the subdomain modeling approach relies on correct reproduction of wet/dry states at and near subdomain boundaries. To ensure this, wet/dry states of subdomain boundary nodes are enforced during the execution of the wetting and drying algorithm twice: (1) right after the execution of the second stage, (2) after all critera are evaluated. Previously, the correctness of wetting and drying in subdomains were verified by a set of logical statements and a final theorem [12]: “Provided the water surface elevations of the subdomain nodes are the same as the corresponding full domain nodes, enforcing both intermediate and final wet/dry states of subdomain boundary nodes using the corresponding final wet/dry states obtained from the full domain run results in the same final wet/dry states of the subdomain grid nodes as the final wet/dry states of the corresponding full domain nodes.”

In this section, as an alternative to the theorem, we use the verification model of the algorithm to prove the correctness of wetting and drying for subdomains. We declare two domains, full and sub, and execute the abstract algorithm sequentially for each domain. The assertion to prove the correctness of wetting and drying for subdomains is that the final wet/dry states of both domains must be equivalent.

The transfer of boundary data from a full domain to a subdomain is done by an array of channels, called bc. Once the execution of the algorithm for a full domain is completed, a
non-deterministic choice of sending boundary conditions, or skipping, is made for all nodes, except the internal node, in inline function `sendBCs()`. If the non-deterministic choice is to send the boundary conditions to a node, then the node is treated as a boundary node in the execution of subdomain, and its wet/dry state is enforced in inline function `receiveBCs()`, called twice. An example configuration of a full domain and a subdomain is shown in Figure 4.24. The declarations of the array of channels and inline functions are shown in Figure 4.25.

![Figure 4.24: An example configuration for abstract full domain and subdomain grids.](image)

The correctness of wetting and drying for subdomains also relies on the assumption that water surface elevations, and so the water column heights, of subdomains are the same as the corresponding full domain nodes. Since the decisions regarding water column heights and water surface elevations are abstracted using non-deterministic choices, a mechanism to synchronize the non-deterministic choices of the full domain and subdomain is modeled. The mechanism consists of an array of channels and an inline function, as shown in Figure 4.26. To ensure that the behavior regarding water heights and velocities are synchronized, the non-deterministic decisions apparent in the model are substituted with the calls to the inline function `determine(bvar,p,i)`, where `bvar` is the boolean variable representing the outcome of

120
// Message channel for b.c. transfer:
chan bc[nn_max] = [2] of {bool};

// Sends b.c. to sub:
inline sendBCs(domain){
    for (i:1..nn-1){
        if bc[i] != isWet(domain, i) -> bc[i] = isWet(domain, i) ->
            printf("node[%d] is subdomain boundary\n",i)
        skip
    fi
}
}

// Receives b.c. from full
inline receiveBCs(domain){
    d_step{
        for (i:1..nn-1){
            if bc[i] ? domain.nodes[i].tempWet -> bc[i] ? domain.nodes[i].tempWet
                else -> skip
        fi
    }
}
}

Figure 4.25: The declarations of the boundary conditions channel and inlines.

the decision, p is the index of the control point, and i is the index of the node or element. When the function is called, it is first checked in the conditional statement if a decision is made before, at the control point p, for the component i, by checking whether channel ndc[p] has a message stored for the component i, using the statement:

ndc[p] ?? [bvar, eval(i)]

If the statement is executable, i.e., if there is a message stored in the channel for the component i, the value of the message is copied to bvar, and therefore, the decisions of both domains are synchronized. If there is not a message stored, i.e., if a decision at p for the component i is not made before, bvar is set non-deterministically, and the decision is stored in the channel for future use.

An example usage of this mechanism can be seen in the modified inline function part3(domain), as seen in Figure 4.27. Note that, since the non-deterministic choice is outside the conditional
#define nndc 6 // no. of ctrl pts at which the decisions are to be synced

chan ndc[nnndc] = [nn_max] of {bool, byte}

inline determine(bvar, p, i) {
    if :: ndc[p] ?? [bvar, eval(i)] -> ndc[p] ?? bvar, eval(i)
    else if :: bvar = true;
    :: bvar = false;
    fi
    ndc[p] ! bvar, i;
    fi
}

Figure 4.26: A mechanism for non-deterministic choice synchronization.

Statement, d_step is used to further increase the performance of the model.

inline part3(domain) {
    printf("Part -3: Elemental Drying\n");
    byte h_ge_hoff;
    for (i:0..ne-1) {
        if :: nWetNodes(domain, i)<2 -> makeEleDry(domain, i);
        :: else ->
            determine(h_ge_hoff, 3, i);
            d_step{
                if :: h_ge_hoff -> printf(" h.ge.hoff for element %d\n", i)
                :: else -> printf(" h.lt.hoff for element %d (barely wet nodes)\n", i) ->
                    domain.elements[i].wet = false;
                    printf(" element[%d] is made wet\n", i);
                fi
            }
        fi
    }
}

Figure 4.27: The modified inline of the third part of the algorithm.

The final modification to the model is to add the effects of uncertainties resulting from the changes in the adjacencies of subdomain boundary nodes. Recall that the effects of external nodes on the behavior of internal nodes are accommodated using non-deterministic choices. In
order to eliminate over-approximation of subdomain models which may cause false positives [92], non-deterministic choices for the effects of external nodes must be refined so that the decisions made by both domains may actually be realized in ADCIRC. For example, if a subdomain boundary node is determined to be landlocked in the full domain, the corresponding node in the subdomain must also be landlocked, because if all the elements connected to the node outside the verification model in the full domain are inactive, then it must also be the case for the subdomain. If, however, the node is not landlocked in the full domain because it is connected to an active element outside the verification model, the corresponding subdomain boundary node may or may not be connected to the corresponding active element, and therefore, the choice is non-deterministic. To accommodate this behavior, the nodal loop in `part4()` is modified as shown in Figure 4.28.

```c
// Landlocking routine for the interface nodes:
for (i:1..nn-1){
    if
        :: (isTempWet(domain,i) && !(domain.nodes[i].mju) && !(isSubBdry(i))) ->
            determine(landlocked,4,i);
        d_step{
            if
                :: landlocked -> printf("nodes[%d] is landlocked\n",i) ->
                    makeTempDry(domain,i);
                :: else -> skip;
            fi
        }
        :: (isTempWet(domain,i) && !(domain.nodes[i].mju) && (bc[i] ? [landlocked])
        )
            if
                :: landlocked -> printf("nodes[%d] is landlocked\n",i) -> makeTempDry( domain,i);
            :: else if
                :: landlocked -> printf("nodes[%d] is landlocked\n",i) ->
                    makeTempDry(domain,i);
                :: skip;
            fi
            fi
        :: else -> skip;
    fi
}
```

Figure 4.28: The modified nodal loop in `part4(domain)`. 

123
Modeling an additional domain doubles the number of properties and exponentially increases the state space. Thus, as suggested in the outlined modeling approach, we make an additional abstraction to reduce the size of state space. Since the states that would result from the changes made in `part1()` are already included in the set of initial states, and since the modifications for subdomain modeling approach interfere only after `part2()`, the first part of the algorithm is opted out. Once modifications to the inline functions are completed, the initial process including the assertion for the correctness of wetting and drying of subdomains is modeled as shown in Figure 4.29. The verification model confirms that the assertion is not violated and so the wetting and drying procedure for subdomains is accurate.

To confirm that intermediate and final forcings of wetting and drying in subdomains are necessary, we comment out calls to `receiveBCs()` at the intermediate stage and at the final stage, respectively. Each change results in assertion violations, confirming that both forcings are indeed necessary. The shortest path to the counterexample found in the absence of intermediate forcing, for example, is shown in Figure 4.30, and progression of wet/dry states of nodes and elements are shown in Figure 4.31. The interpretation of the execution trail is:

The subdomain boundary node, `node[3]`, is initialized dry but made wet in `part2` of the full domain. Since the full domain element causing the node to be made wet is outside the subdomain mesh, the node stays dry in subdomain execution. The boundary node, `node[3]`, staying erroneously dry, causes elements 1 and 2 to become inactive, which results in erroneous landlocking of the internal node; `node[0]`.

### 4.6 Limitations

As in any other application of model checkers, verification models of discrete numerical models suffer from state-space explosion problem. An effective abstraction of these models plays a critical role in preventing an excessive size of state space that makes the analysis of the model intractable. Abstraction of such systems must be implemented carefully and any redundancy
in the model must be avoided, which may be challenging in some cases. A recent overview of techniques for avoiding state space explosion problem is presented by Groote et al. [48]. A discussion of how to avoid redundancy in SPIN models can be found in the SPIN model checker primer and reference manual [56].

Another limitation of the outlined verification approach is that the mesh abstraction is applicable only to properties with a finite information propagation speed, i.e., a finite domain of dependence. For instance, contrary to numerical models of hyperbolic PDEs, a numerical
Number of nodes: 4

Full Domain
Initializing nodes
node[0] initialized wet
node[1] initialized wet
node[2] initialized dry
node[3] initialized dry
Part-2: Nodal Wetting
vss.le.vmin
node 3 made wet
Part-3: Elemental Drying
h.lt.hoff for element 0 (barely wet nodes)
  element[0] is made wet
h.ge.hoff for element 2
Executing Part-4: Nodal Drying

Subdomain
Initializing nodes
node[0] initialized wet
node[1] initialized wet
node[2] initialized dry
node[3] initialized dry
node[1] is subdomain boundary
node[3] is subdomain boundary
Part-2: Nodal Wetting
vss.le.vmin
Part-3: Elemental Drying
h.lt.hoff for element 0 (barely wet nodes)
  element[0] is made wet
h.ge.hoff for element 2
Executing Part-4: Nodal Drying
nodes[0] is landlocked

Analysis
Comparison for node 0, full:1, sub:0
spin: wetdry-sub.pml:313, Error: assertion violated

Figure 4.30: The shortest path to the counterexample found in absence of intermediate forcing.
(The trail includes additional printf statements for further clarification.)

model of a parabolic PDE, e.g., heat equation, has an infinite propagation speed, i.e., an infinite
domain of dependence [69], and therefore is inapplicable for our verification modeling approach
because the behavior of a node depends on all other nodes of the mesh at a timestep.
Figure 4.31: Progression of wet/dry states of nodes and elements in absence of intermediate wet/dry forcings for the subdomain. Dry nodes and elements are shown in yellow, while wet nodes and elements are shown in blue. Inactive elements are either dry or has at least one dry node.
4.7 Related Work

Model checking tools have been utilized by scientific software developers for addressing the issues associated with parallelism such as equivalence of parallel and serial versions of a program, race conditions, deadlocks, and correctness of floating-point operations. Siegel et al. [87], for example, combine model checking with symbolic execution to verify the correctness of parallel numerical programs, by using a sequential version as a specification for the parallel version. In a subsequent study, Siegel et al. [88] present a verification suite for scientific computing applications to show the functional equivalence of sequential and parallel versions of an algorithm. Similarly, Burnin et al. [25] propose an approach to verify a parallel program by using its sequential version with controlled non-determinism. Boldo and Nguyen [21] present an approach to proving numerical programs which may have inconsistencies in floating-point arithmetics due to different system architectures. Pervez et al. [83] demonstrate the applicability of model checking for high-performance programs and argue that concurrency-related errors, such as deadlocks and race conditions can easily be detected using model checking tools. Arnold et al. [8] present a new method to verify sparse matrix codes by specifying sparse codes as functional programs and by using Isabelle/HOL, a proof assistant for higher-order logic, to verify the full functional correctness of such programs.

4.8 Conclusions

Wetting and drying schemes in hydrodynamic models behave algorithmically complex and unpredictable. The complexity of the algorithmic behavior increases further when additional features are to be added. While testing falls short of detecting all potential issues and exceptions associated with model configuration and the algorithm, deductive reasoning requires time and expertise in logical reasoning.

In this study, we propose the utilization of model checking tools to abstract discrete model systems and to verify their critical properties. We outline an abstraction and verification ap-
proach, and model ADCIRC’s wetting and drying algorithm. We present two test cases that were previously encountered by ADCIRC modelers. The first test case is the detection of an undesired behavior in the algorithm, which was originally diagnosed through test runs. The second test case is the verification of the correctness of the algorithm for child domains of the adaptive subdomain modeling approach. The verification approach successfully detects the issues and verifies the algorithmic behavior with more confidence (due to the exhaustive state-space exploration strategy) and using much less computational effort and time. We, therefore, conclude that model checking tools can effectively be used to verify the algorithmic behavior of numerical ocean models, especially when new components and techniques are incorporated.

Our future directions include utilizing SPIN and other model checking tools for the analysis and verification of similar decisional algorithms and routines of numerical models of physical phenomena.


[19] CA Blain, JJ Westerink, and RA Luetich. The influence of domain size on the response


Appendix A

ADCIRC++ Input File Formats

Table A.1: Format of ADCIRC++ project input file, where header is an alphanumeric description of the input file, projectID is an alphanumeric id for the project, nd is the number of domains, domainID_i, configDir_i, and outputPath_i are the domain id, configuration file directory, and path to the output files of the i.th domain, respectively, and parentID_i is the id of the parent domain of i.th domain. (If the domain i is a parent domain, the fourth column in line 3+i is left blank.

<table>
<thead>
<tr>
<th></th>
<th>header</th>
<th>projectID</th>
<th>nd</th>
</tr>
</thead>
<tbody>
<tr>
<td>3+i</td>
<td>domainID_i, configDir_i, outputPath_i [parentID_i]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A.2: Format of ADCIRC++ domain configuration file, where header is an alphanumeric description of the input file, domainType is the type of the domain (=“ADCIRC” for ADCIRC domains), type_i is the type of the i.th input file (=14 for grid files, =15 for model configuration files, etc.), format_i is the format of the i.th input file (=“ADCIRC” for ADCIRC domains), fileDir_i is the directory of the i.th input file relative to the directory of the configuration file.

<table>
<thead>
<tr>
<th></th>
<th>header</th>
<th>domainType</th>
</tr>
</thead>
<tbody>
<tr>
<td>2+i</td>
<td>type_i, format_i, fileDir_i</td>
<td></td>
</tr>
</tbody>
</table>
Table A.3: Format of ADCIRC++ .dif file, where \( np \) is the number of properties to be locally modified, propertyID\( _i \) is the id of \( i \).th property to be modified (e.g., “bathymetry”, “mannings-n_at_sea_floor”, etc.), \( nn \_i \) is the number of nodes with modified propertyID\( _i \), \( n\_i\_j \) and \( val\_i\_j \) are the node number and the new value of the modified property \( i \) of the \( j \).th node in the set of nodes with modified propertyID\( _i \).

\[
\begin{array}{c|c}
1 & np \\
\hline
l\_i & for i in np: \\
\hline
l\_i+1 & propertyID\_i \\
\hline
l\_i+1+j & for j in nn\_i: \\
\hline
l\_i+1+j & n\_i\_j \ val\_i\_j \\
\end{array}
\]

Table A.4: Format of optional ADCIRC++ .asm file, where \texttt{header} is an alphanumeric description of the input file, \( \texttt{tau} \) is the tolerance, \( \texttt{sigma} \) is the safety factor, \( \texttt{lambda} \) is the decay constant, \( \texttt{theta} \) is the minimum activation interval.

\[
\begin{array}{c|c}
1 & \texttt{header} \\
2 & \texttt{tau} \\
3 & \texttt{sigma} \\
4 & \texttt{lambda} \\
5 & \texttt{theta} \\
\end{array}
\]
Appendix B

Auxiliary inline and macro definitions

```cpp
// Sets i.th node of a given domain to be wet
inline makeWet(domain, i) {
    domain.nodes[i].tempWet = true;
    domain.nodes[i].wet = true;
}

// Sets i.th node of a given domain to be dry
inline makeDry(domain, i) {
    domain.nodes[i].tempWet = false;
    domain.nodes[i].wet = false;
}

// Sets i.th node of a given domain to be temp. wet
inline makeTempWet(domain, i) {
    domain.nodes[i].tempWet = true;
}

// Sets i.th node of a given domain to be temp. dry
inline makeTempDry(domain, i) {
    domain.nodes[i].tempWet = false;
}

// Sets i.th element of a given domain to be wet
inline makeEleDry(domain, i) {
    domain.elements[i].wet = false;
}
```

Figure B.1: Auxiliary inline definitions for the abstract wetting and drying algorithm.
#define isWet(domain, i) \
    domain.nodes[i].wet

#define isTempWet(domain, i) \
    domain.nodes[i].tempWet

#define isElementWet(domain, i) \
    domain.elements[i].wet

#define isSubBdry(i) \
    bc[i] ? [...] \

// Returns the number of wet nodes of element i
#define nWetNodes(domain, i) \
    (isWet(domain, n1(i)) + \ 
     isWet(domain, n2(i)) + \ 
     isWet(domain, n3(i)))

// Returns the number of temp. wet nodes of element i
#define nTempWetNodes(domain, i) \
    (isTempWet(domain, n1(i)) + \ 
     isTempWet(domain, n2(i)) + \ 
     isTempWet(domain, n3(i)))

Figure B.2: Auxiliary macro definitions for the abstract wetting and drying algorithm.