Modeling a Discrete Wet-Dry Algorithm for Hurricane Storm Surge in Alloy

John Baugh(✉) and Alper Altuntas

Civil, Construction, and Environmental Engineering,
North Carolina State University, Raleigh, NC, USA
{jwb,aaltunt}@ncsu.edu

Abstract. We describe an Alloy model that helps check the correctness of a discrete wet-dry algorithm used in a system for hurricane storm surge prediction. Derived from simplified physics and encoded with empirical rules, the algorithm operates on a finite element mesh to allow the propagation of overland flows. Our study is motivated by complex interactions between the algorithm and a recent performance enhancement to the system that involves mesh partitioning. We briefly outline our approach and describe safety properties of the extension, as well as directions for future work.

1 Introduction

The tools and techniques most often associated with scientific computing are those of numerical analysis and, for large-scale problems, structured parallelism to improve performance while limiting complexity. Beyond those conventional tools, we also happen to see a role for state-based methods and present one such application here using the Alloy language [4]. Our immediate concern is the correctness of an extension made by our group to ADCIRC [5], a finite element code widely used by the U.S. Army Corps of Engineers and others to simulate storm surge. ADCIRC itself has been extensively validated against actual flooding conditions, with simulation times of about 1 000 CPU hours.

To get a sense of the problem, the mesh in Fig. 1 depicts a shoreline extracted from a larger domain with 620 089 nodes and 1 224 714 triangular elements that encompasses the western North Atlantic Ocean, the Caribbean Sea, and the Gulf of Mexico. Forced with winds and tides, three primary routines drive the physics of the model and are executed in succession at each time step. The first finds the free surface elevation for each node in the domain. Next the wet-dry status of each node is determined via a series of checks involving water surface elevation, velocity, and prior wet-dry states. Finally, velocities at each node are determined by solving the shallow water equations.

Our extension, now included in ADCIRC, is an exact reanalysis technique that enables the assessment of local *subdomain* changes with less computational effort than would be required by a complete resimulation [1]. Figure 2 shows a domain Ω partitioned at interface Γ into a subdomain Ω_I, representing the

M. Butler et al. (Eds.): ABZ 2016, LNCS 9675, pp. 256–261, 2016.
DOI: 10.1007/978-3-319-33600-8_18

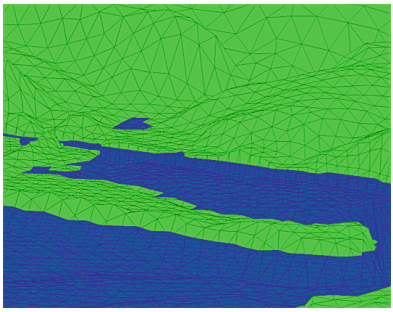

Fig. 1. Finite element mesh

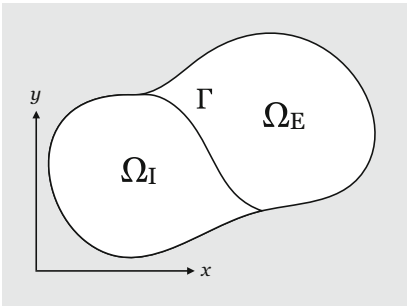

Fig. 2. Region of interest Ω_I

interior of a geographic region of interest, and Ω_E. The technique starts with a simulation on Ω that produces elevations, velocities, and wet-dry states that are used as *boundary conditions* along interface Γ in subsequent low-cost simulations on Ω_I. We refer to the first simulation as a *full run* and the latter as a *subdomain run*. A correctness condition requires that boundary conditions be enforced in such a way that results obtained in both cases match within subdomain Ω_I.

A particularly tricky interface condition arises from ADCIRC's discrete wet-dry algorithm, which operates on a finite element mesh to accommodate advancing and receding flood waters [3]. We begin by describing a spatial representation in Alloy that forms the basis for state used by the algorithm.

2 Statics: Representing a Mesh

Finite element methods work by discretizing a continuous domain and approximating a solution with piecewise polynomials. The resulting mesh of elements and nodes can be thought of as a triangulation of a surface. We begin with a representation of mesh topology and later add physical attributes:

```
sig Mesh {
   triangles: some Triangle
}
abstract sig Vertex {}
```

```
abstract sig Triangle {
   edges: Vertex → Vertex,
   adj: set Triangle
}
```

Facts are defined to ensure that every triangle has three directed edges and is oriented, i.e., its edge set forms a ring. Distinct triangles with common anti-parallel edges define the *adj* relation and, correspondingly, the dual of a mesh:

```
fact { all t, t': Triangle | t in t'.adj iff one ~(t.edges) & t'.edges }
```

Using *adj*, we ensure that a mesh is connected and oriented; edges are required to be unique. Other facts prevent the possibility of local "cut points" in the mesh as well as overlapping triangles, with the latter following from Euler's formula, $T - E + V = 1$, where T triangles, E (undirected) edges, and V vertices exist. These are specified in terms of helper functions and predicates that define and distinguish between interior and border vertices and edges.

Algorithm 1. Wetting and Drying

0: **for** e **in** *elements* **do** ▷ start with all elements being wet
  make e wet
1: **for** n **in** *nodes* **do** ▷ make nodes with low water column height dry
  if W_n **and** $H_n < H_0$ **then**
    $W_n \leftarrow$ false, $W_n^t \leftarrow$ false
2: **for** e **in** *elements* **do** ▷ propagate wetting
  if e has exactly 2 wet nodes **and** $V_{ss}(e) > V_{min}$ **then** unless slow flow
    Let j be the remaining dry node | $W_j^t \leftarrow$ true
3: **for** e **in** *elements* **do** ▷ allow water to build up
  find nodes i and j of e with highest water surface on downhill slopes
  if H_i, $H_j < 1.2H_0$ **then**
    make element e dry
4: **for** n **in** *nodes* **do** ▷ make landlocked nodes dry
  if W_n^t **and** n on only inactive elements **then**
    $W_n^t \leftarrow false$
5: **for** n **in** *nodes* **do** ▷ set the final wet-dry state for nodes
  $W_n \leftarrow W_n^t$

3 Dynamics: Wetting and Drying

The purpose of the wet-dry routine, characterized in Algorithm 1, is to determine which nodes participate in the calculation of physical properties in the next time step. Its output is W_n for each node n, which is set true when the node is wet. Within the algorithm, both nodes and elements have intermediate wet-dry states, determined by current physical properties, that are set and unset, e.g., W_n^t, which is true when a node is "temporarily" wet. Additionally, as part of the algorithm, an element is said to be *active* if it is wet and has three temporarily wet nodes, and a node is *landlocked* if it is incident only to inactive elements.

The algorithm has both spatial and temporal dimensions, with the former being maintained by vertices and triangles that are now extended to include features of the problem domain:

```
sig Node extends Vertex {
  W, Wt: Bool → State,
  H: one Height
}
sig State {}
```

```
sig Element extends Triangle {
  wet: Bool → State,
  slowFlow: one Bool,
  lowNode: one Node
}
```

In addition to discrete wet-dry states $n.W$ and $n.Wt$ for node n, and $e.wet$ for element e, we incorporate physical attributes and tests on them by making use of predicate abstraction: a water column height $n.H$ may be low ($H_n < H_0$), medium ($H_0 \leq H_n < 1.2H_0$), or high ($H_n \geq 1.2H_0$), as used in parts 1 and 3; flow across an element $e.slowFlow$ is true when $V_{ss}(e) \leq V_{min}$, as used in part 2; and $e.lowNode$ is an element's node with the *lowest* water surface, supporting

the test in part 3 of the algorithm. To accommodate local state changes within a mesh, a *State* atom is added in the last column of the *W*, *Wt*, and *wet* relations.

We allow the algorithm to begin with arbitrary *n.W* states, as though they had been produced in a prior time step, and check correctness at the end. Each part of the algorithm is modeled by a predicate defining the state change:

```
pred part2 [s, s': State] {
  noElementChange[s, s']
  all n: Node | n.W.s' = n.W.s
    and (make_wet[n, s] implies n.Wt.s' = True else n.Wt.s' = n.Wt.s) }
```

where *noElementChange*[*s*, *s'*] specifies the frame condition, and *make_wet*[*n*, *s*] defines the conditions in part 2 that cause a node to become wet, namely:

```
pred make_wet [n: Node, s: State] {
  some e: Element | e.slowFlow = False and loneDryNode[n, e, s] }

pred loneDryNode [n: Node, e: Element, s: State] {
  n in dom[e.edges] and n.W.s = False and wetNodes[e, s] = 2 }

fun wetNodes [e: Element, s: State]: Int {
  #(dom[e.edges] <: W).s.True }
```

Other parts of the algorithm are similarly defined.

4 Full and Subdomain Runs

With the above, we are able to represent a mesh and the dynamic behavior of ADCIRC's wet-dry algorithm by chaining together its parts and thereby constraining intermediate states to form a trace. What is left is to distinguish between full and subdomain runs (denoted *F* and *S*), which we achieve by extending *State* so that a unique trace can be generated for each type of run:

```
sig F, S extends State {}
```

Then, by making use of predicate abstraction, we can have a single mesh instance do double duty and serve the needs of both. Here is how. We recognize that, within Ω_I, the two types of runs perform identical computations, though on their own state variables. Where the two cases differ is along interface Γ.

For a full run, we represent only Ω_I and use nondeterminism along Γ to model arbitrary behavior external to it, as depicted in Fig. 3 (in white). Boundary nodes are defined to realize that capability:

```
sig Boundary extends Node {
  allowsWetting, allowsDrying: one Bool }
```

where *n.allowsWetting* is true when a node n on Γ in a full run is incident to an *imaginary* element e in Ω_E that has exactly two wet nodes and $V_{ss}(e) > V_{min}$, and *n.allowsDrying* is likewise true when such an element is active.

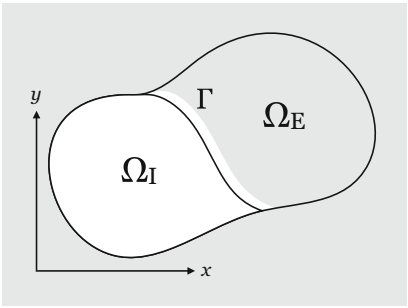

Fig. 3. Mesh for Full Run

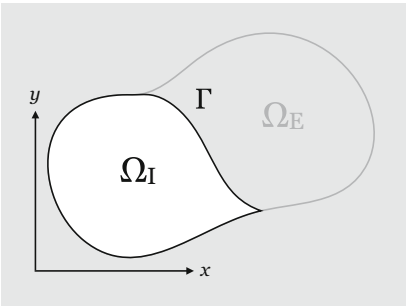

Fig. 4. Mesh for Subdomain Run

The conditions defined by the two fields can then be used in parts 2 and 4 of the algorithm, respectively, to account for interactions with Ω_E in a full run. Within part 2, for instance, *make_wet* now becomes:

```
pred make_wet [n: Node, s: State] {
  (some e: Element | e.slowFlow = False and loneDryNode[n, e, s])
  or (s in F and n in Boundary and n.allowsWetting = True) }
```

Intuitively, the updated predicate makes clear that subdomains require state from a prior full run on Γ if they are to produce final wet states that match their full domain counterparts.

Naturally, for a subdomain run, we represent only the portion of the domain over which an ADCIRC simulation is performed, i.e., the geographic region of interest, Ω_I, as shown in Fig. 4 (in white). Absent any forcing along the interface, results internal to it clearly diverge from those produced by a full run. The following assertion confirms this by producing a counterexample:

```
assert sameFinalStates {
  all n: Node | n.W.FD/last = n.W.SD/last }
```

where *FD* is an ordering on F, and *SD* is an ordering on S, so $n.W.FD/last$ denotes the final wet state of a node n in a full run, for instance.

4.1 Enforcing Boundary Conditions

For actual simulations in ADCIRC, we can store the intermediate states of boundary nodes produced during a full run, and then use them as boundary conditions in a subdomain run. Doing so makes subsequent low-cost simulations possible, since all computations external to a geographic region of interest, Ω_I, are avoided. In practice, that cost is only a fraction of a percent of the time required for full runs [1].

The value of state-based modeling in Alloy is in gaining confidence that the boundary conditions are right, since it facilitates experimentation with (a) the amount of state along interface Γ needed from a full run, and (b) the manner in

which that state is enforced in subdomain runs. To satisfy the *sameFinalStates* assertion, for instance, we pull wet-dry states out of a full run and apply them to a subdomain run. This modification is made to the last conjunct in the *part*2 predicate—call it x—so that it becomes:

(s **in** S **and** n **in** Boundary **implies** n.Wt.s' = n.W.FD/last **else** x)

With a similar change in part 4, we are able to show that enforcing *intermediate* wet-dry states on subdomain boundary nodes with the corresponding *final* wet-dry states obtained from a full run is sufficient to satisfy safety properties. Thus, for actual simulations in ADCIRC, we can record a minimal amount of data from a full run—the final wet-dry states on Γ—and during a subdomain run force those states in parts 2 and 4 of the wet-dry algorithm.

5 Conclusion and Future Work

As far as we are aware, ours is the first study to model in Alloy some of the discrete computational aspects of a finite element solver for systems of partial differential equations. Among several related studies, however, are one that uses Larch and CCS on an illustrative numerical algorithm [2] and another that combines symbolic execution and model checking for numerical subroutines [6].

Our look at model checking began with a question ADCIRC developers raised but were unable to answer without resorting to experiments: how is it that an element with three wet nodes can apparently be dry? We quickly put together Promela/SPIN and FSP/LTSA models that produced traces from a limited set of constructively-defined topologies. A strength of Alloy, of course, is model *generation,* allowing mesh topologies to be defined by declarative properties instead of trying to devise an algorithm to produce them. Given the prevalence of network-like structures of various types in science and engineering, we imagine that Alloy might find further uses there. Our own efforts are focused on a reimplementation of ADCIRC that incorporates adaptivity both for reanalysis and mesh refinement. We expect to make use of Alloy's support for experimenting with abstractions, building object models, and finding representation invariants.

References

1. Baugh, J., et al.: An exact reanalysis technique for storm surge and tides in a geographic region of interest. Coast. Eng. **97**, 60–77 (2015)
2. Chadha, H., Baugh, J., Wing, J.: Formal specification of concurrent systems. Adv. Eng. Softw. **30**, 211–224 (1999)
3. Dietrich, J.C., Kolar, R.L., Luettich, R.A.: Assessment of ADCIRC's wetting and drying algorithm. Dev. Water Sci. **55**, 1767–1778 (2004)
4. Jackson, D.: Software Abstractions: Logic, Language, and Analysis (revised edition). MIT Press, Cambridge (2012)
5. Luettich, R.A., Westerink, J.J.: Formulation and Numerical Implementation of the 2D/3D ADCIRC Finite Element Model Version 44.xx. http://www.adcirc.org
6. Siegel, S., et al.: Combining symbolic execution with model checking to verify parallel numerical programs. ACM TOSEM **17**(2), 1–34 (2008). Article no. 10