Science of Computer Programming 158 (2018) 100–121

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Formal methods and finite element analysis of hurricane storm surge: A case study in software verification

John Baugh *, Alper Altuntas

Department of Civil, Construction, and Environmental Engineering, North Carolina State University, Raleigh, NC, USA

ARTICLE INFO

Article history:
Received 7 October 2016
Received in revised form 18 August 2017
Accepted 21 August 2017
Available online 8 September 2017

Keywords:
Formal methods
Model checking
Scientific computing
Earth and atmospheric sciences

ABSTRACT

Used to predict the effects of hurricane storm surge, ocean circulation models are essential tools for evacuation planning, vulnerability assessment, and infrastructure design. Implemented as numerical solvers that operate on large-scale datasets, these models determine the geographic extent and severity of coastal floods and other impacts. In this study, we look at an ocean circulation model used in production and an extension made to it that offers substantial performance gains. To explore implementation choices and ensure soundness of the extension, we make use of Alloy, a declarative modeling language with tool support and an automatic form of analysis performed within a bounded scope using a SAT solver. Abstractions for relevant parts of the ocean circulation model are presented, including the physical representation of land and seafloor surfaces as a finite element mesh, and an algorithm operating on it that allows for the propagation of overland flows. The approach allows us to draw useful conclusions about implementation choices and guarantees about the extension, in particular that it is equivalence preserving.

1. Introduction

Coastal flooding from tropical storms is the result of large-scale processes whose simulation is computationally demanding. Ocean circulation models, which attempt to capture the resulting hydrodynamics as accurately and efficiently as possible, employ a variety of mechanisms that, while improving performance, can also lead to complex software implementations. Based on numerical techniques such as finite element methods [41], these models start from a characterization of wind velocities, atmospheric pressure, and land and seafloor surfaces to produce time histories of spatially varying water surface elevations and velocities. Ocean circulation models are designed to perform these large-scale simulations while incorporating both tidal effects and more extreme storm events into the analysis.

As in other areas of science, validation of the models is based on observational comparisons and, as much as anything, review and evaluation by an active community of scientists and engineers. Quantities like wind, wave, and water level are compared with data from the National Data Buoy Center (NDBC), National Ocean Service (NOS) observation stations, high water marks, and other sources [23]. In addition to such studies, since 2008 a regional testbed has been actively comparing the accuracy of ocean circulation models and their relative abilities to hindcast the effects of historical storms [33].

* Corresponding author.
E-mail address: jwb@ncsu.edu (J. Baugh).

http://dx.doi.org/10.1016/j.scico.2017.08.012
0167-6423/

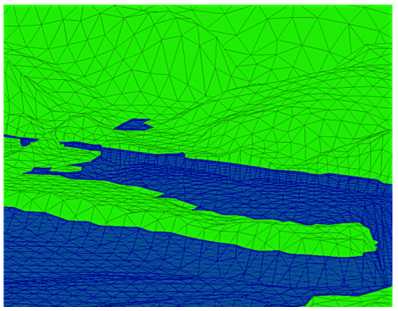


Fig. 1. Finite element mesh.

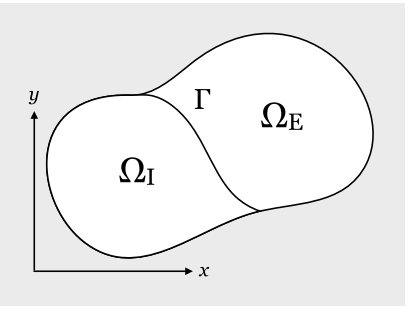


Fig. 2. Domain Ω partitioned at interface Γ.

In this paper, our concern is the correctness of an extension made by our group to ADCIRC [29], an ocean circulation model widely used by the U.S. Army Corps of Engineers and others to simulate hurricane storm surge. ADCIRC itself has been extensively validated against actual flooding conditions, with simulation times of about a thousand or more CPU-hours.

To get a sense of the problem, the finite element *mesh* in Fig. 1 depicts a shoreline extracted from a larger domain that encompasses the western North Atlantic Ocean, the Caribbean Sea, and the Gulf of Mexico. The land and seafloor surfaces are represented as a collection of contiguous, non-overlapping triangles, or *elements*, that meet along their edges and at their vertices, or *nodes*. For this problem, 620 089 nodes and 1 224 714 elements appear in the full mesh. Forced with winds and tides, the physics of the model are realized in three primary routines that are executed in succession at each discrete time step. The first finds the water surface elevations for nodes in the domain using principles of momentum and continuity. Next, the wet–dry state of nodes is determined from empirical rules so that advancing and receding flood waters can be modeled. Finally, velocities are determined using momentum principles, completing one time step.

Our extension, now included in ADCIRC, is an exact re-analysis technique that enables the assessment of local *subdomain* changes with less computational effort than would be required by a complete resimulation [7]. Fig. 2 shows a domain Ω partitioned at interface Γ into a subdomain Ω_I, representing the interior of a geographic region of interest, and Ω_E. The technique starts with a simulation on Ω that produces water surface 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 on Ω as a *full run*, and the latter one on Ω_I as a *subdomain run*. A correctness condition requires that boundary conditions be imposed in such a way that full and subdomain runs produce equivalent results in region Ω_I.

1.1. Formal methods and verification

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. Beyond those conventional tools, we also see a role for formal methods and present one such application here using the Alloy language and analyzer [25].

Alloy combines first-order logic with relational calculus and associated quantifiers and operators, along with transitive closure. It offers rich data modeling features based on class-like structures and an automatic form of analysis that is performed within a bounded scope using a SAT solver. For *simulation*, the analyzer can be directed to look for instances satisfying a property of interest. For *checking*, it looks for an instance violating an assertion: a counterexample. The approach is *scope complete* in the sense that all cases are checked within user-specified bounds. Alloy's logic supports three distinct styles of expression, that of predicate calculus, navigation expressions, and relational calculus. The language used for modeling is also used for specifying properties of interest and assertions.

Using Alloy, we model finite element domains and simulations on them so we can experiment with the type of boundary conditions that might be imposed on subdomain runs, as illustrated in Fig. 3. The upper left side of the figure represents ADCIRC as it might be employed without our extension: a Fortran implementation and, beneath it, a model of a simulation on Ω, a full domain. The upper right side of the figure represents our extension: a *possible* implementation and, beneath it, a model of a simulation on Ω_I, a subdomain with boundary conditions imposed on Γ. At the bottom of the figure is a comparison between full and subdomain runs in Alloy, where the assertion *SameFinalStates* is checked to see whether they produce equivalent results in region Ω_I. Through an iterative process, we seek an approach for boundary conditions that satisfies the assertion, and then we make use of the insights gained in an *actual* implementation of our extension, which we refer to as *subdomain modeling*.

We observe that other approaches to verification might also be considered. For instance, a mathematical statement of the equations of motion for bodies of water could be adopted as a specification. Because ocean circulation models are defined by a set of hyperbolic partial differential equations (PDEs) [41], however, asking whether an extension such as subdomain modeling implements it requires the tools of numerical analysis in a labor-intensive process that has already been undertaken for ADCIRC. Instead, we utilize the existing foundation and solve a simpler problem: show that the results of a subdomain run are equivalent to a full run in ADCIRC.

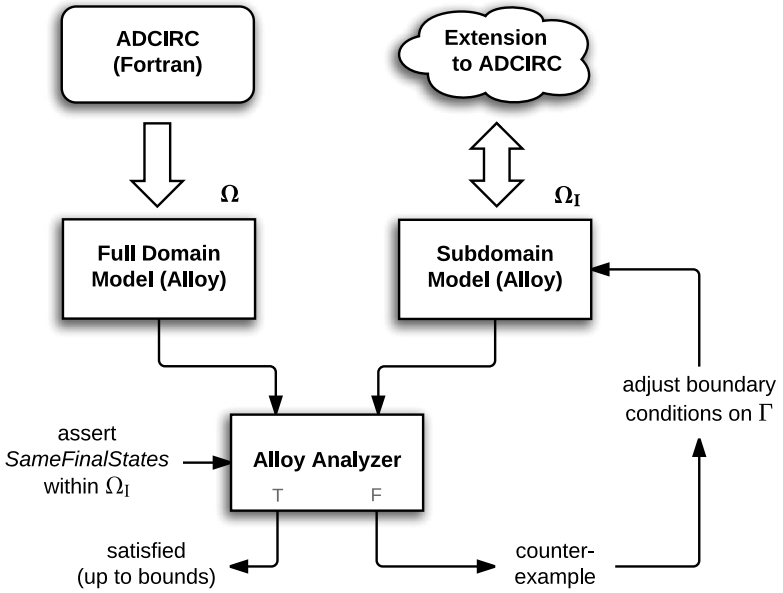


Fig. 3. Verification approach for a *subdomain modeling* extension.

In doing so, we recognize that some aspects of showing equivalence are more challenging to reason about than others, and it is those that receive our attention here. For verifying our subdomain modeling extension, these include (a) the variety of mesh topologies that one may encounter, (b) the process of determining the wet–dry state of nodes in a mesh, and (c) the effects of partitioning a mesh at an interface and imposing boundary conditions. What we adopt, then, is a *lightweight* approach to formal methods [27], in which there is both *partiality in modeling*, since we do not attempt to verify an entire finite element analysis system, and *partiality in analysis*, since the verification we perform is bounded. An additional aspect of lightweight formal methods is an incremental style of modeling, which Alloy supports by offering immediate feedback while models are being constructed. Our presentation follows the same incremental style we used in developing our models.

1.2. Organization of the paper

Originally outlined in a short conference paper [6] at ABZ 2016, our work is extended here to provide additional background on storm surge modeling, an introduction to Alloy language features as they are encountered, more facets of our modeling and verification approach, and examples of using the analyzer for simulation and checking. We also present a new approach for working with and comparing finite element meshes and an additional application of the model. For reference, the Alloy models presented herein are available online [1].

The current paper is organized as follows. Section 2 provides background on ADCIRC and our subdomain modeling extension, along with examples of their use in production and research. Section 3 presents a static model of a finite element mesh that can accommodate simulations without being tied to a particular mesh topology. Section 4 extends the model to include dynamic operations that determine wet–dry state using transition relations and an idiom employed to accommodate local state changes [25]. Section 5 models full and subdomain runs, and shows how boundary conditions can be imposed to achieve equivalence. Section 6 describes an alternative use of Alloy to gain insights into the behavior of empirical algorithms that can be difficult to find by simulation alone. Sections 7 and 8 close the paper with a discussion of related work and conclusions.

2. Background

To provide context for wet–dry computations and subdomain modeling, we begin by describing the discrete aspects of large-scale physical models that make tools like Alloy a potentially useful and complementary part of a computational scientist's toolkit. We also include representative applications of ADCIRC and subdomain modeling as they are used in production and research.

2.1. ADCIRC's wetting and drying algorithm

The subject matter of scientific programs often concerns the physical processes that surround us, where space and time are traditionally viewed as being continuous. Circulation of currents within the atmosphere and oceans, for instance, involves

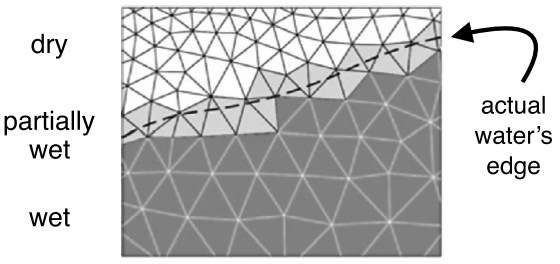


Fig. 4. Mesh illustrating wetting front in reality and as seen by a numerical model [30].

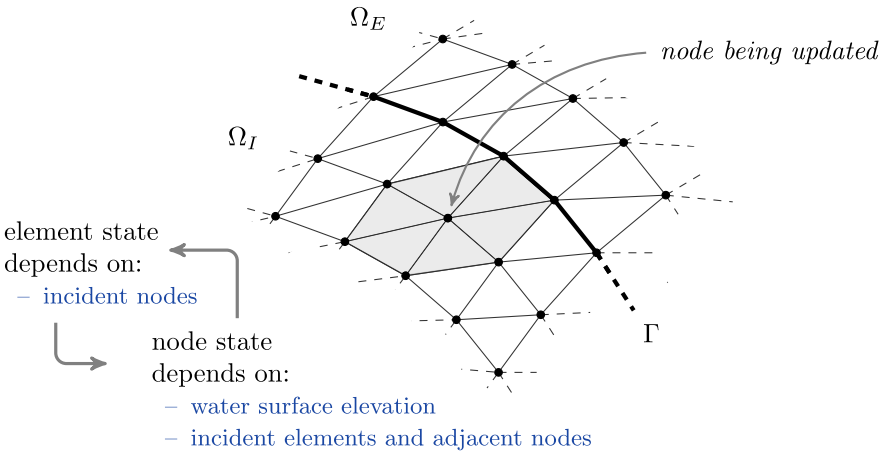


Fig. 5. Dependencies in calculating wet–dry states.

state that is continuously varying and where, indeed, continuity arguments are used to "fill in" gaps that may be associated with sampled data.

The computational apparatus underlying ocean circulation models, however, looks less like purely analytic functions and more like an amalgam of discrete and continuous constructions that allow, as one example, the representation of irregular land and seafloor geometries as piecewise planar surfaces. The types of discretizations that may be employed in both time and space are varied, and each has its own performance, accuracy, and ease-of-development implications.

That problems such as these are discretized implies there is a level beneath which certain physical features and processes are not resolved—a modeling concern to be managed. But more than that, discretizing time and space means that contingent processes, like the continuous movement of the shoreline due to tides and storms—known as *wetting and drying*—require special handling to account for them in physically meaningful ways. To do so, ADCIRC uses an empirical algorithm [30] based on simplified physics to determine whether an element is wet, dry, or "partially" wet, as illustrated in Fig. 4. As a basis for that assessment, the algorithm first allows nodes to be wetted and dried, with wetting determined by a simple momentum balance and drying by a specified minimum water depth. Because of problems with mass balance and instabilities in regions with steep slopes, however, subsequent improvements in the algorithm have been made to address them and thereby prevent unrealistic consequences, such as the uninterrupted flow of a thin film of water [12,13].

In terms of implementation, ADCIRC's wet–dry algorithm produces a series of state changes in nodes and elements at each time step before final wet–dry states can be determined. Fig. 5 illustrates topological dependencies that, combined with those state changes, complicate reasoning about the wet–dry algorithm and its interplay with mesh partitioning. Nodes internal and external to a subdomain are shown, as well as interface nodes that must be treated specially during a subdomain run and forced with data obtained from a full run: a boundary condition.

2.2. Subdomain modeling

Before a hurricane threatens a local community, it forms over the deep ocean, moves across the continental shelf, and pushes water onto the coastal floodplains. To accommodate this large-scale behavior, model domains typically span sizable parts of the globe, as the finite element mesh in Fig. 6 illustrates. Because the computational cost of even a single simulation can be high, and because separate runs are needed for every scenario considered, finding a way to reduce the cost of repeated runs is important.

Subdomain modeling relies on the observation that local changes to a mesh in a geographic region of interest have only local effects at the relevant time scales. As a result, for any change under consideration, a subdomain can be defined that

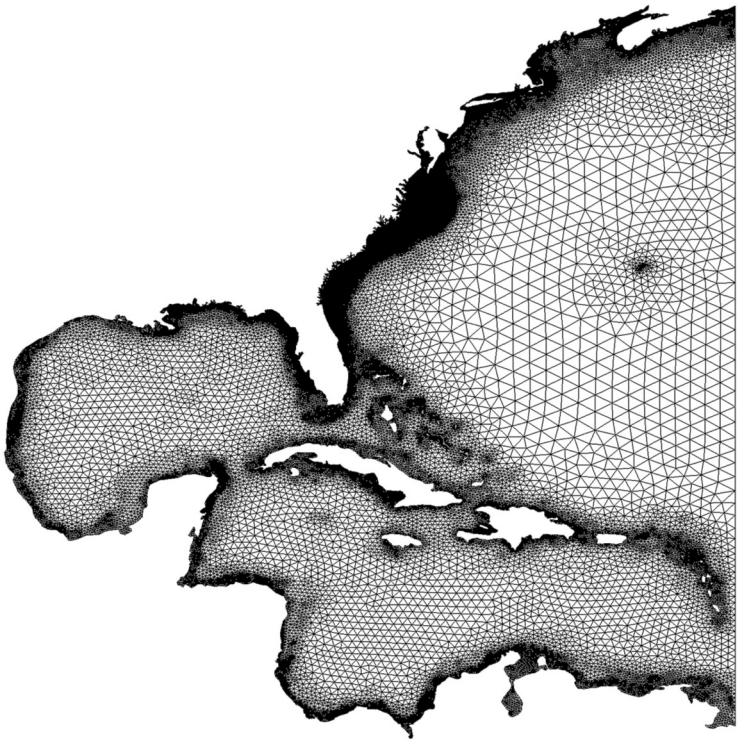
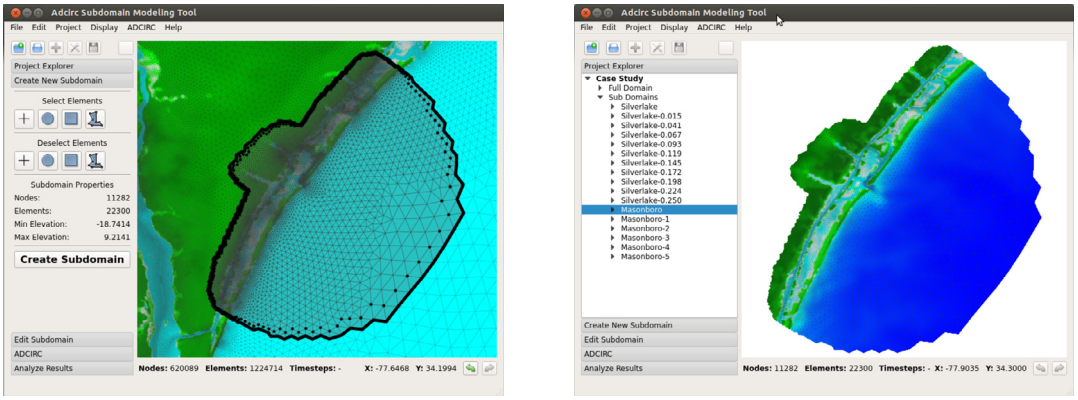


Fig. 6. Typical domain covering the Western North Atlantic, Caribbean Sea, and the Gulf of Mexico.



(a) Selecting subdomain nodes and elements (b) Extracting a region of interest

Fig. 7. Application of the Subdomain Modeling Tool [14] along the North Carolina Coast.

encompasses the area of influence of the change, together with boundary conditions determined from a prior simulation of the entire domain. Doing so allows us to analyze the effects of an incremental change at an incremental computational cost. Figs. 7a and 7b show the selection and extraction, respectively, of a subdomain along the North Carolina coast using the Subdomain Modeling Tool [14].

The extraction of a subdomain requires mesh partitioning and the imposition of boundary conditions, which may be formulated in a variety of ways. To be effective, the output obtained in full and subdomain runs must be equivalent within the geographic region of interest. By output, we mean the basic quantities produced by ADCIRC, i.e., water surface elevations, velocities, and wet–dry states at each node and for every time step. The nature of the computations is such that, of the three quantities, enforcing wet–dry states on boundaries is the most challenging and the one we tackle here.

2.3. Applications of ADCIRC and subdomain modeling

Uses of ADCIRC in production include operations, planning, and design activities by the U.S. Army Corps of Engineers (USACE), Federal Emergency Management Agency (FEMA), National Oceanic and Atmospheric Administration (NOAA), U.S. Coast Guard (USCG), and other agencies and engineering firms. Some prominent examples include the design of the $14.5 billion Hurricane and Storm Damage Risk Reduction System (HSDRRS) in Southern Louisiana by USACE that was completed in 2011 [36–38], the development of next-generation flood risk maps by FEMA to establish actuarial flood insurance rates and promote sound floodplain management [16], and assistance in making storm-related decisions by USCG that affect deployment locations and continuity of operations.

Subdomain modeling was incorporated in ADCIRC v51.42 in early 2015 and in experimental versions years ahead of that. In one of its earliest applications, subdomain modeling was used in post-Katrina studies by USACE's Flood and Storm Protection Division, where the technique "yielded considerable time and cost savings in the calculations" when applied as part of an investigation to examine the effect of pumping discharge from the Western Closure Complex on storm surge levels along the downstream communities. The Western Closure Complex is a key component of the hurricane storm damage reduction system for New Orleans.

In other studies, subdomain modeling has been used to determine surface roughness coefficients that measure resistance to the flow of water as part of a formulation of a stochastic inverse problem [21]. Because of the computational expense in doing so, Graham [20] reduces the cost of a series of forward models using subdomain modeling in the Hurricane Gustav case study from about 3 300 to 11 CPU-hours. The authors note that "the work would not have been possible" without subdomain modeling. In another study, Haddad et al. [22] investigate factors affecting the behavior of storm surge in wetlands by combining field work and numerical modeling. While not reporting specific performance gains, the authors state that subdomain modeling can reduce the computational cost of repeated simulations that require adjustments to the mesh and the vegetation parameters in the regions of interest.

3. Statics: representing a mesh

Finite element methods [41] work by discretizing a continuous domain and approximating a solution over it with piecewise polynomial functions. The resulting mesh of elements and nodes can be thought of as a triangulation of a surface, with every node located in three-dimensional space.

While spatial attributes and physical quantities play a role in ADCIRC's wet–dry algorithm, we separate concerns here and begin with a representation of mesh topology alone—with *triangles* and *vertices* as basic components—that we later embellish with other attributes, as dictated by the algorithm.

3.1. Mesh as a planar triangulation

A planar triangulation can be defined in Alloy as follows for arbitrary mesh topologies:

```
sig Mesh {
    triangles: some Triangle,
    adj: Triangle → Triangle
}
```

```
sig Triangle {
    edges: Vertex → Vertex
}
sig Vertex {}
```

A *signature* in Alloy introduces both a type and a set of uninterpreted atoms, along with *fields* that define relations over them. In addition to defining a type, a signature's name can also be used within an Alloy expression to denote the set of elements it defines.

In the declarations above, the signature *Mesh* has a field *triangles* that defines a binary relation over *Mesh* and *Triangle*; the multiplicity keyword *some* indicates that *triangles* associates at least one triangle with each mesh.[1] A mesh m containing triangle t, then, would be represented by a tuple (m, t) in *triangles*. An additional field *adj* defines a ternary relation—using Alloy's arrow operator—that contains the tuple (m, s, t) when a mesh m includes "adjacent" triangles s and t. Similarly, in signature *Triangle*, the field *edges* is a relation that contains the tuple (t, u, v) when a triangle t includes a directed edge from vertex u to v. The *Vertex* signature, on the other hand, contains no fields and therefore defines no additional relations. The sets *Mesh*, *Triangle*, and *Vertex* are referred to as *top level* and are therefore disjoint.

By way of example, a mesh with three triangles and five vertices that is consistent with the declarations above is shown in Fig. 8. In Alloy, sets are represented as unary relations, so a set of *Triangle* atoms appears as $\{(t_0), (t_1), (t_2)\}$. Scalars are just a special case and are represented by unary relations with only one tuple. This *instance* and others are produced by the Alloy analyzer, allowing a user to see whether a given assignment of values to variables matches his or her intent. In this case, a meaningful interpretation can be given to the instance, which has the planar embedding shown in Fig. 9, with triangles that are defined by their directed edges, and edges that are defined by their incident vertices. The Alloy analyzer

[1] In a signature **sig** S { f: e }, if e is an expression denoting a set, it has a default multiplicity of *one*. In other words, **sig** S { f: e } and **sig** S { f: **one** e } are equivalent. Without the *some* keyword, then, the *triangles* field would associate just one triangle (exactly one) with each mesh.

$$
\begin{aligned}
\text{Mesh} &= \{(m_0)\} \\
\text{Triangle} &= \{(t_0), (t_1), (t_2)\} \\
\text{Vertex} &= \{(v_0), (v_1), (v_2), (v_3), (v_4)\} \\
\text{triangles} &= \{(m_0, t_0), (m_0, t_1), (m_0, t_2)\} \\
\text{adj} &= \{(m_0, t_0, t_1), (m_0, t_1, t_0), \\
&\quad (m_0, t_1, t_2), (m_0, t_2, t_1)\} \\
\text{edges} &= \{(t_0, v_0, v_2), (t_0, v_2, v_1), (t_0, v_1, v_0), \\
&\quad (t_1, v_1, v_2), (t_1, v_2, v_3), (t_1, v_3, v_1), \\
&\quad (t_2, v_3, v_2), (t_2, v_2, v_4), (t_2, v_4, v_3)\}
\end{aligned}
$$

Fig. 8. An instance of a mesh in Alloy.

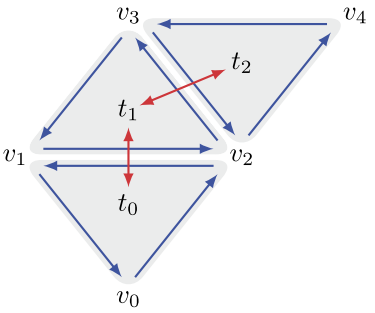


Fig. 9. A planar embedding of the instance as a mesh of triangles (t_0, t_1, and t_2), vertices (v_0, v_1, v_2, v_3, and v_4), and edges (in blue), with triangle adjacency (in red), cf. Figs. 8 and 10.

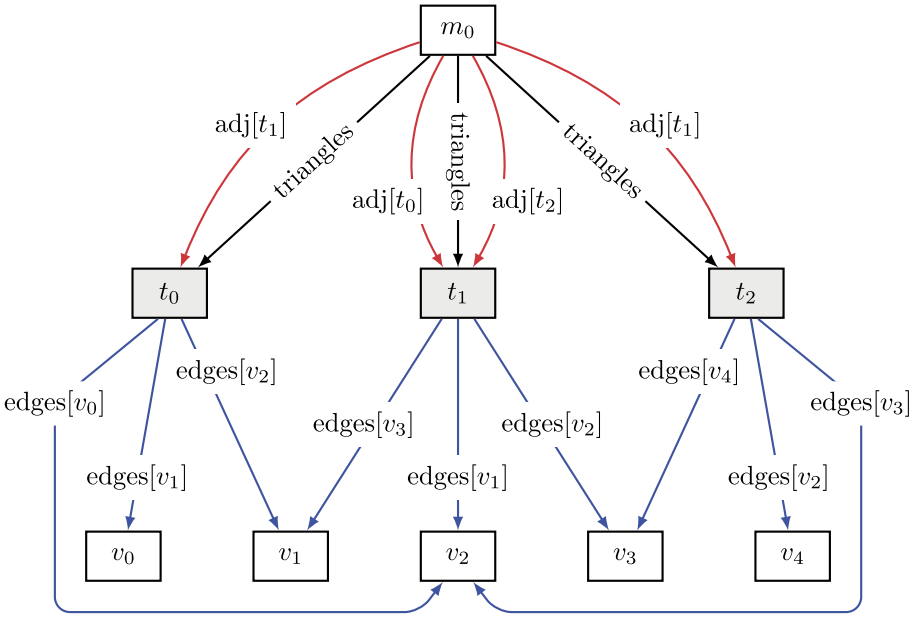


Fig. 10. The same instance displayed as a graph, with arcs that represent relations between atoms.

also provides a visual representation of the instance as a graph, with arcs that represent relations between atoms, as shown in Fig. 10, where the box notation ([ ]) used on arcs *adj* and *edges* is defined below.

Among Alloy's operators is a generalized form of relational composition, *dot join*, which takes on its usual meaning in the expression $p.q$ when p and q are binary relations. When p is a set or scalar, $p.q$ is relational image, mirroring the navigation expressions commonly found in object oriented languages. So, for instance,

$$m_0.\text{adj} = \{(t_0, t_1), (t_1, t_0), (t_1, t_2), (t_2, t_1)\}$$

where the scalar m_0 is treated the same as $\{m_0\}$, (m_0), and $\{(m_0)\}$: all are represented as $\{(m_0)\}$ in Alloy. Differing only syntactically from the dot operator is *box join*, where $e_1[e_2]$ has the same meaning as $e_2.e_1$ for expressions e_1 and e_2, though with different precedence. So

$$m_0.\text{adj}[t_1] = t_1.(m_0.\text{adj}) = \{(t_0), (t_2)\}$$

That is, in mesh m_0, the triangles adjacent to t_1 are t_0 and t_2. In general, we require for any mesh m that *m.adj* be symmetric, irreflexive, and strongly connected.

3.2. Adding constraints

Because other instances produced by Alloy may satisfy the signatures without having a valid interpretation, we must say more precisely what we mean by a mesh, and we do so using *facts* that constrain the possible assignments of values to variables. For instance, we require every triangle to have three (directed) edges with the universally quantified constraint:

```
fact { all t: Triangle | #t.edges = 3 }
```

where *t.edges* denotes the vertex pairs of triangle t, and where $\#r$ is the number of tuples in relation r.

Another basic requirement is that, for a mesh m, the adjacency relation *m.adj* is defined over its own set of triangles, *m.triangles*:

```
fact { all m: Mesh | dom[m.adj] + ran[m.adj] in m.triangles }
```

where *dom*[r] and *ran*[r] are the domain and range, respectively, of a binary relation r, and where $+$ is set union.

For convenience in defining and working with meshes, we make use of their *orientability*, a property derived from their application to land and seafloor surfaces. When decomposing a surface into triangles, each can be oriented in a direction around its perimeter so that the edges of adjacent triangles point in opposite directions, as we saw in Fig. 9. A triangle's edge set therefore forms a *ring*, which we can ensure by requiring (a) that each of its vertices has exactly one successor and (b) that all of them are reachable from any vertex by following edges repeatedly:

```
fact { all t: Triangle | ring[t.edges] }

pred ring [e: Vertex → Vertex] {
   all v: dom[e] | one v.e and dom[e] in v.^e
}
```

The term $v.e$ is the set of successors of v, and the quantifier *one* requires that there be exactly one such successor for each vertex v. The term $v.\hat{}e$ denotes the vertices reachable from v using the transitive closure $\hat{}e$. Facts, which constitute constraints in Alloy that must always hold, can make use of predicates like *ring*, which are simply a way of packaging expressions for use by one or more facts.

Because triangles are oriented, we know that each of the edges in a mesh will be unique if we are to avoid overlapping triangles. That is, no mesh should include both (t, u, v) and (t', u, v) in *edges* unless $t = t'$, or:

```
fact { all m: Mesh | all disj t, t': m.triangles | no t.edges & t'.edges }
```

where *disj* makes t and t' distinct, and where & is set intersection. Triangles with common *anti-parallel* edges define the *adj* relation and, correspondingly, the dual of a mesh:

```
fact { all m: Mesh, t, t': m.triangles | t in m.adj[t'] iff one ~(t.edges) & t'.edges }
```

where $\sim r$ is the transpose of binary relation r.

Intuitively, triangles are like basic building blocks of a mesh that can be glued together edge-to-edge when the edges are anti-parallel. The relation *m.adj* allows us to define the kind of connectivity we expect of a mesh m:

```
fact { all m: Mesh |
          let r = m.adj, s = m.triangles |
             symmetric[r] and irreflexive[r] and stronglyConnected[r, s] }
```

where *symmetric*, *irreflexive*, and *stronglyConnected* operate on binary relations and are provided by modules included in the Alloy library.

An additional constraint is necessary to ensure that a mesh is physically meaningful, in particular that it has a planar embedding. As employed by ocean modelers, a mesh is the product of a triangulation of a set of points in the plane, so it should therefore satisfy Euler's formula for a simple polygon, whose characteristic equals one. By convention, such polygons are defined in terms of *undirected* graphs of vertices, edges, and faces that are, in our case, triangular. Thus, we require that $V - E + T = 1$, where V vertices, E (undirected) edges, and T triangles appear in a mesh. The mesh in Fig. 9, for instance, satisfies the formula, since $V = 5$, $E = 7$, and $T = 3$, with E being determined by ignoring edge direction and merging any common pairs in the obvious way.

In our representation, the number of undirected edges E can be found by subtracting half the number of *interior* edges, i.e., those that have an anti-parallel mate, from the total number of edges in a mesh:

```
fun undirectedEdges [m: Mesh]: Int {
   let e = m.triangles.edges | minus[#e, div[#(~e & e), 2]]
}
```

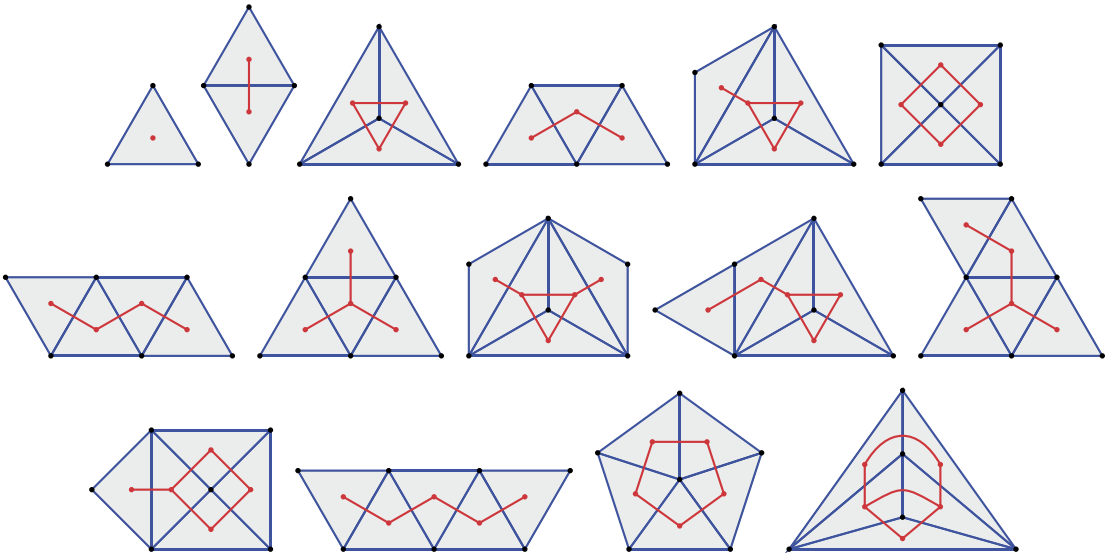


Fig. 11. A menagerie of mesh topologies (up to $T = 5$).

where *m.triangles.edges* denotes all edges in mesh *m*, and where ~*e* & *e* denotes just the interior ones. So, for the mesh in Fig. 9 we have $E = 9 - 4/2 = 7$ since there are 9 total edges and 4 interior ones: (v_1, v_2), (v_2, v_1), (v_2, v_3), and (v_3, v_2). From the figure, it should be clear that the number of tuples in *m.adj* always equals the number of interior edges.

Euler's formula can then be expressed as follows:

```
fact { all m: Mesh |
         let T = #m.triangles, E = undirectedEdges[m], V = #dom[m.triangles.edges] |
           minus[T, 1] = minus[E, V] }
```

With our definition of a mesh complete, the Alloy analyzer produces the topologies shown in Fig. 11, where triangle edges are in blue and adjacency is in red; both are drawn without directions to keep the figures simple. For T triangles, all M distinct topologies are represented, with $(T, M) = (1, 1), (2, 1), (3, 2), (4, 4)$, and $(5, 7)$.

3.3. A simple check

Before moving on, one property we might like to check in Alloy is that our representation prevents the possibility of local "cut points," where the effective width of a mesh is zero, i.e., where connectivity is maintained only by a single vertex. To do so, we define the two types of vertex that are valid in a mesh: those on the border of a mesh and those in the interior. Both can be characterized by the number of edges without anti-parallel mates that are incident on them, i.e., two in the case of border vertices, and zero for interior vertices:

```
pred borderVertex [m: Mesh, v: Vertex] {
  let e = m.triangles.edges | #symDiff[e.v, v.e] = 2
}
pred interiorVertex [m: Mesh, v: Vertex] {
  let e = m.triangles.edges | no symDiff[e.v, v.e]
}
```

where *m.triangle.edges.v* is the set of vertices directed *to* v, *v.(m.triangle.edges)* is the set of vertices directed *from* v, and *symDiff*[*a*, *b*] is the symmetric difference of sets *a* and *b*.

For instance, vertex v_3 in Fig. 9 happens to be a border vertex, as do all the others in this mesh: the set of vertices directed to v_3 are $\{(v_2), (v_4)\}$ and from v_3 are $\{(v_1), (v_2)\}$, and their symmetric difference is $\{(v_1), (v_4)\}$, meaning there are two edges without anti-parallel mates that are incident on v_3, so it is indeed a border vertex.

Using the Alloy analyzer we can define and check assertions to see if they follow from the stated signatures and facts. Here, we state that every vertex must be either a border vertex or an interior one:

```
assert NoCutPoints { all m: Mesh, v: Vertex | borderVertex[m, v] or interiorVertex[m, v] }
```

noting that a cut point, by way of contrast, would have exactly *four* incident edges without anti-parallel mates. Because Alloy performs bounded verification, we must specify a scope for the analysis. To keep run times brief, we choose at most

Algorithm 1 Wetting and drying.

```
0: for e in elements do                                   ▷ initialization: start with all elements being wet
       wet_e ← true
1: for n in nodes do                                      ▷ make nodes with low water column height dry
       if W_n and H_n < H_min then
           W_n ← false, W_n^t ← false
2: for e in elements do                                   ▷ propagate wetting across element
       if ¬W_i for exactly one node i on e and V_ss(e) > V_min then   unless flow is slow
           W_i^t ← true
3: for e in elements do                                   ▷ allow water to build up
       find nodes i and j of e with highest water surface elevations η_i and η_j   on an incline
       if min(H_i, H_j) < 1.2H_min then
           wet_e ← false
4: for n in nodes do                                      ▷ make landlocked nodes dry
       if W_n^t and n on only inactive elements then
           W_n^t ← false
5: for n in nodes do                                      ▷ set the final wet–dry state for nodes
       W_n ← W_n^t
```

6 triangles and 9 vertices, and Alloy finds and guarantees that there are no counterexamples within that scope in under a minute on a laptop computer with a 2.8 GHz Intel Core i7.

4. Dynamics: wetting and drying

Packaged as a module, the mesh representation given can now serve as an abstraction for finite element analysis and other computations that work on planar triangulations, where its components—triangles and vertices—are given interpretations appropriate for the intended application domain. In the case of the wet–dry algorithm, we extend triangles and vertices with some but not all of the attributes that appear in ADCIRC itself, as needed.

In ADCIRC, a *node* can be viewed as a vertex located in a three-dimensional Euclidean space with x, y, z coordinates and with quantities like water surface elevation, velocity, and wet–dry state that vary with time, among others. An *element* can be viewed as a triangle in a plane defined by its incident nodes, along with its own wet–dry state and physical quantities that vary in time. To model the wet–dry routine, a node's location in three-space is not required, as we show, but what *is* required is a combination of mesh topology and a set of predicates over some of those physical quantities.

We begin by describing the wet–dry algorithm in an imperative style using pseudocode, which follows its expression in ADCIRC, albeit in an abstract manner. We then extend our Alloy model and specify the algorithm's behavior declaratively using predicates that relate before and after states of its individual parts. Some simple simulations are then performed to gain confidence in the fidelity of the specification.

4.1. The wet–dry algorithm

Called at each time step, the wet–dry algorithm is a set of empirical rules that operate on a finite element mesh to determine which nodes participate in the calculation of physical properties in the next time step. Defined by Algorithm 1, it produces as output W_n for each node n and wet_e for each element e, Boolean variables that are set true when the associated component is considered "wet." Taken together, W_n and wet_e determine whether or not there are numerical contributions in ADCIRC to the equations that calculate water surface elevations and velocities. Comments accompanying each part of the algorithm offer a glimpse at the simplified physics upon which the rules are based.

Looking at individual parts of the algorithm, we see that nodes have additional wet–dry states that are set and unset. These are denoted by W_n^t, which is true when a node n is considered "temporarily wet." Additionally, as part of the algorithm, an element is said to be *active* if and only if it is wet and has three temporarily wet nodes, and a node is *landlocked* if and only if it is incident only to inactive elements.

Though both nodes and elements have wet–dry states that are potentially of interest, only those pertaining to nodes are affected—and complicated—by mesh partitioning, the problem we are addressing. The wet–dry states of elements, on the other hand, are determined solely by physical quantities that are known to be correct through other, more straightforward means, as presented elsewhere [7]. So while they appear in our model of the algorithm, we may safely focus our attention on W_n values as the algorithm's output, which are compared in full and subdomain runs as a measure of correctness of the boundary conditions imposed on subdomains.

4.2. Extending triangles and vertices

Before modeling individual parts of the algorithm, we first extend triangles and vertices to include relevant aspects of the problem domain:

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

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

Subtype signatures in Alloy introduce no new types but instead represent sets of elements that are subsets of their parents. Through fields, they may also introduce additional relations. Here, the new signatures introduce a set *Node* as a subset of *Vertex*, a set *Element* as a subset of *Triangle*, and a new type and set *State*. Additional aspects of the declarations warrant comment, including notation, how state changes are accommodated, and how physical quantities determined by other parts of ADCIRC are incorporated.

With respect to notation, we define discrete wet–dry states that mirror those in the wet–dry algorithm, so elements have a *wet* field, and nodes have *W* and *Wt* fields for wet and temporarily wet, respectively. To accommodate local state changes within a mesh, a *State* atom is added in the last position of those relations. So when node n is dry in state 4 and then becomes wet in state 5, the relation W contains the following tuples: (n, *False*, $State_4$) and (n, *True*, $State_5$). As a result of the approach, a simple join expression $n.W.s$ denotes the value of the relation W for node n at a given state in the algorithm. We could have just as well swapped the order of *Bool* and *State* in the fields so that *State* appears in the middle of the relation, but doing so would be less notationally convenient. Use of the multiplicity keyword *one* indicates that we require *n.W.s* to denote a single Boolean state, as we also require of the *Wt* and *wet* fields.

Adding the *State* signature in the declarations of *wet*, *W*, and *Wt* fields, as we have done, allows them to "vary" in the sense that they become a function of *State*, an idiom commonly employed in Alloy to make a relation dynamic [25]. Note by way of contrast that the *State* signature does not appear in the declaration of the *H*, *slowFlow*, and *lowNode* fields, since those represent physical quantities that are static for the duration of a time step in the wet–dry algorithm, as we describe below. Note also that the fields in *Mesh* and *Triangle* that we saw earlier are static in this sense: they do not have a changing state, since mesh topology is also static.

In addition to discrete wet–dry states, physical attributes are incorporated by modeling the tests performed on them using predicate abstraction [19], resulting in a slight over-approximation of the original algorithm, as we discuss in Section 6.1.

For a node n, a water column height $n.H$ represents the vertical distance at that location from the water surface to the ocean floor, as used in parts 1 and 3 of the algorithm. It may be low ($H_n < H_{min}$), medium ($H_{min} \leq H_n < 1.2H_{min}$), or high ($H_n \geq 1.2H_{min}$), where H_{min} is a user-defined constant:

```
abstract sig Height {}
one sig Low, Med, High extends Height {}
```

An abstract signature like *Height* has no elements other than the ones defined by its extensions, and *one sig* declares signatures *Low*, *Med*, and *High* to be sets containing exactly one element.

For an element e, a flow of water *e.slowFlow* is true when $V_{ss}(e) \leq V_{min}$, as used in part 2 of the algorithm. V_{ss} is the steady-state velocity resulting from differences in water surface elevations of an element's incident nodes. It is a function of those elevations, the distance between nodes, and their bottom friction; nodal velocities are not used. V_{min} is a user-defined constant. The value *e.lowNode* is the element's node with the *lowest* water surface elevation—or one of the lowest in the case it is not unique—supporting the test in part 3 of the algorithm. It must of course be a node incident to the element, so we have:

```
fact { all e: Element | e.lowNode in dom[e.edges] }
```

The entire collection of signature declarations, to this point, can be represented graphically as a model diagram, as shown in Fig. 12, which the Alloy analyzer itself is able to generate from a textual description. In the figure, boxes represent sets of atoms, and two different types of arrowhead are used to distinguish between relations (thin, filled arrowheads) and subtype relationships (fat, unfilled arrowheads).

Finally, because the wet–dry algorithm iterates over collections of nodes and elements, we make use of the following functions for convenience and clarity:

```
fun nodes [m: Mesh]: set Node { dom[m.triangles.edges] }
fun elements [m: Mesh]: set Element { m.triangles }
```

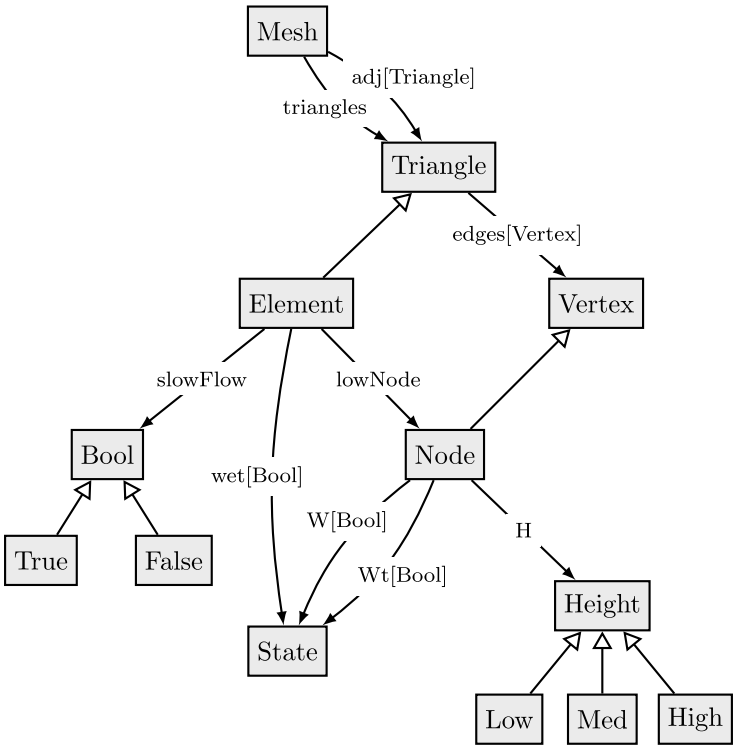


Fig. 12. Model diagram: a graphical representation of the model's declarations.

4.3. The algorithm as transition relations

Equipped with the above, we now turn to individual parts of Algorithm 1, where each is modeled by a transition relation: a predicate defining the change in state it produces. We begin by allowing the algorithm to start from arbitrary *n.W* states, as though they had been produced in a prior time step, and then observe their values at the end.

Initialization of the algorithm appears below, where elements are defined to be wet, as prescribed by the algorithm, and nodes are allowed to be either wet or dry, so long as their wet and temporarily wet states match (again, as they would at the end of a prior time step):

```
pred init [m: Mesh, s: State] {
    all e: m.elements | e.wet.s = True
    all n: m.nodes | n.W.s = n.Wt.s
}
```

We note here, as a matter of syntax, that a block defined by braces ({}) in Alloy may enclose a sequence of constraints; its meaning is equivalent to the conjunction of those constraints.

Now, taking each of the parts in turn, nodal drying is performed in part 1, causing nodes with low water column height to be made dry:

```
pred part1 [m: Mesh, s, s': State] {
    noElementChange[m, s, s']
    all n: m.nodes |
        n.W.s = True and n.H = Low
            implies n.W.s' = False and n.Wt.s' = False
            else n.W.s' = n.W.s and n.Wt.s' = n.Wt.s
}
```

where the conditional statement in part 1 of the algorithm is modeled with logical implication, the test on water column height appears as *n.H = Low*, and the predicate *noElementChange*[*m*, *s*, *s'*] specifies the frame condition, since an element's wet–dry state is left unchanged in this part:

```
pred noElementChange [m: Mesh, s, s': State] {
    all e: m.elements | e.wet.s = e.wet.s'
}
```

In part 2, elements with a single dry node cause that node to become wet if the steady-state velocity across the element is sufficient. Though an imperative implementation of this would loop over elements, as in Algorithm 1, a more declarative description requires instead that we begin with nodes and identify those that are incident to such elements:

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

where *make_wet*[*m*, *n*, *s*] defines the conditions that cause a node to become wet, namely:

```
pred make_wet [m: Mesh, n: Node, s: State] {
    some e: m.elements | 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
}
```

For a node *n* to become wet, it must be incident to an element whose steady-state water velocity is sufficient, as determined by term *e.slowFlow*, and with a single node that is dry, as determined by the *loneDryNode* predicate. Function *wetNodes* counts the nodes of the element in which *W* is true in the given state; in its definition, the form $s <: r$ is domain restriction of r to s, which in this case is the relation *W* restricted to the nodes incident on *e*.

In part 3, the algorithm allows water to build up in "barely wet" elements to prevent the uninterrupted flow of a thin film of water, as noted in Section 2.1:

```
pred part3 [m: Mesh, s, s': State] {
    noNodeChange[m, s, s']
    all e: m.elements |
        let ij = dom[e.edges] − e.lowNode |                    -- nodes with highest water surface elevations
            e.wet.s' = (some ij.H − High implies False else e.wet.s)
}
```

where a let expression introduces the variable *ij*, which is bound to the two incident nodes *i* and *j* with the highest water surface elevations, where the expression $ij.H - High$ is non-empty if either *i* or *j* has a water column height of *Low* or *Med*, and where the predicate *noNodeChange*[*m*, *s*, *s'*] specifies the frame condition:

```
pred noNodeChange [m: Mesh, s, s': State] {
    all n: m.nodes | n.W.s = n.W.s' and n.Wt.s = n.Wt.s'
}
```

In part 4, nodes that are landlocked, i.e., surrounded by inactive elements, are made temporarily dry:

```
pred part4 [m: Mesh, s, s': State] {
    noElementChange[m, s, s']
    all n: m.nodes {
        n.W.s' = n.W.s
        n.Wt.s' = (make_dry[m, n, s] implies False else n.Wt.s)
    }
}
```

where *make_dry*[*m*, *n*, *s*] defines the conditions that cause a node to become dry, namely:

```
pred make_dry [m: Mesh, n: Node, s: State] {
    n.Wt.s = True and landlocked[m, n, s]
}
pred landlocked [m, Mesh, n: Node, s: State] {
    no { e: m.elements | n in dom[e.edges] and active[e, s] }
}
pred active [e: Element, s: State] {
    e.wet.s = True and all n: dom[e.edges] | n.Wt.s = True
}
```

where an element is *active* if and only if it is wet and all three of its nodes are temporarily wet.

In part 5, the final wet–dry states for nodes are assigned from their temporary values:

```
pred part5 [m: Mesh, s, s': State] {
    noElementChange[m, s, s']
    all n: m.nodes | n.W.s' = n.Wt.s and n.Wt.s' = n.Wt.s
}
```

The complete wet–dry algorithm can now be expressed as follows, with parts being combined together using sequential composition:

```
pred solve [m: Mesh, s: Int → State] {
    init[m, s[0]]
    part1[m, s[0], s[1]]
    part2[m, s[1], s[2]]
    part3[m, s[2], s[3]]
    part4[m, s[3], s[4]]
    part5[m, s[4], s[5]]
}
```

where five transitions are performed over six states (0..5) that are collected together in a mapping *s* from integer indices to states.

Looking ahead to the simulations performed below, when visualizing instances in Alloy, it is helpful to have state atoms appear in lexical order, which we can ensure by importing Alloy's built-in ordering module:

```
open util/ordering [State] as so
```

where the *open* statement includes both the path and the name of the module, and because it is parameterized, an instantiation type. An optional alias *so* is used to distinguish between different instantiations, if and as needed. Here, the *ordering* module imposes a total ordering of the atoms of State, whose names are interchangeable, so there is no loss of generality. The module provides functions that can be accessed via the alias we chose, e.g., *so/first*, *so/next*, and *so/last*, for working with ordered states.

4.4. Some simple simulations

Our approach to verification involves a comparison between full and subdomain runs, as opposed to showing that the wet–dry algorithm, in isolation, satisfies a specification. Before setting up the comparison, we can perform some simple simulations of the algorithm to gain confidence that it behaves as expected.

For instance, to see how a finite element mesh can start out with all wet nodes that then become dry in a single time step—i.e., in one execution of the wet–dry algorithm—we define the following predicate and look for satisfying instances:

```
pred allWetToDry {
    let s = toSeq[so/first, so/next] |
        some m: Mesh |
            solve[m, s] and all n: m.nodes | n.W.(s[0]) = True and n.W.(s[5]) = False
}
```

where *toSeq* is a helper function that takes an ordering of states, as described above, and produces a mapping from indices to states. The initial and final states are denoted by *s*[0] and *s*[5], respectively, though we could have just as well used *so/first* and *so/last*.

An arbitrarily large number of instances of arbitrary size are produced by Alloy, showing that all nodes can indeed start out wet and become dry if, for instance, they all happen to have low water column heights (**all** *n*: *m.nodes* | $n.H = Low$),

which will cause them to become dry in part 1. Further wetting in part 2 is then impossible, since it would require the propagation of wetting from *some* other (wet) nodes. In actual simulations with ADCIRC, such an outcome is unlikely since it would be prevented by mass conservation requirements external to the wet–dry algorithm.

Alternatively, by defining an analogous predicate *allDryToWet*, we can ask whether a mesh can start out with all dry nodes that then become all wet. No such instances are produced, as expected, since a dry node can only become wet through the propagation of wetting from elsewhere, which is accomplished in part 2 of the algorithm.

5. Full and subdomain runs

To accommodate both full and subdomain runs simultaneously, we begin by describing the structural relationships between them, and then show how to set up comparisons so that outcomes can be used to determine whether proposed boundary conditions on subdomains are effective. Several different approaches for boundary conditions are described as part of the iterative process introduced in Section 1.1 and illustrated in Fig. 3. We also present an alternative modeling approach and highlight some of its own characteristics.

In terms of representation, the structure is simple enough: a subdomain covers part of a full domain, as we saw in Fig. 2, where domain Ω is partitioned at interface Γ into a region of interest, Ω_I, and a region external to it, Ω_E. We extend *Mesh* with the singletons *Full* and *Sub* to represent Ω and Ω_I, respectively, and then require that the subdomain's elements be drawn from those of the full domain:

```
one sig Full, Sub extends Mesh {}
fact { all e: Sub.elements | e in Full.elements }
```

where, as with any other mesh, the elements in *Sub* are required to be contiguous. With respect to coverage, the subdomain may have either no, some, or all borders in common with the full domain. In the latter case, for instance, the subdomain and full domain are equivalent, and we expect the states produced by the wet–dry algorithm to be equivalent even without imposing boundary conditions on interface Γ.

By sharing structure in this manner, the domains have common properties, such as the physical attributes *slowFlow* and *H*, which correspond to steady-state velocity and water column height, respectively. These denote computations in both full domains and subdomains that are external to the wet–dry algorithm, and that are taken to be correct and consistent based on the results of previous work [7].

While full and subdomain runs share structure, their individual computations should be independent and based on their own wet–dry states. To distinguish between them, we extend *State* so that a unique trace is generated for each type of run:

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

where *F* and *S* correspond to full and subdomain runs, respectively.

If runs are compared at this point, absent any special handling of nodes along interface Γ, the results produced by a subdomain run diverge from those produced by a full run. The assertion *SameFinalStates* sets up this case by initializing runs so that they begin in the same wet–dry states (at $f[0]$ and $s[0]$), and by then checking their final states (at $f[5]$ and $s[5]$) for equivalence:

```
assert SameFinalStates {
  let f = toSeq[fo/first, fo/next], s = toSeq[so/first, so/next] |
    { all n: Sub.nodes | n.W.(f[0]) = n.W.(s[0])      -- start from the same wet–dry states
      solve[Full, f]
      solve[Sub, s]
    } implies all n: Sub.nodes | n.W.(f[5]) = n.W.(s[5])
}
```

where, as before, the aliases *fo* and *so* refer to a total ordering of the atoms that define full domain and subdomain states, respectively, and where f and s refer to the corresponding sequences.

As expected, the assertion produces counterexamples, and does so with as few as a single element in Ω_E, the region outside of subdomain Ω_I, and with one or more elements within the subdomain. Each counterexample produced falls into at least one of two broad categories: one where wetting would have entered the subdomain from Ω_E but does not, and another where drying occurs that would have been prevented by an active element in Ω_E. In the subdomain run, since region Ω_E is not part of the simulation, these behaviors are absent. Obviously, such behaviors can and do occur in many different ways and combinations.

5.1. Imposing boundary conditions

For actual simulations in ADCIRC, we can store wet–dry states on interface Γ during a full run and then impose them on subsequent subdomain runs. Doing so makes low-cost simulations possible, since all computations external to a geographic

region of interest can then be avoided. In practice, that cost is only a fraction of a percent of the time required for full runs [7].

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 Γ needed from a full run, and (b) the manner in which that state is enforced in subdomain runs, as we show.

In defining boundary conditions, we have several choices that could involve either nodes or elements, or both, along with their intermediate and final wet–dry states. We start by identifying nodes on interface Γ with the following predicate:

```
pred gamma [m: Mesh, n: Node] {
    m = Sub and borderVertex[m, n] and some incidentElts[Full, n] − incidentElts[Sub, n]
}
fun incidentElts[m: Mesh, n: Node]: set Element {
    { e: m.elements | n in dom[e.edges] }
}
```

where *gamma* is true of a node if it appears on the border of a subdomain and if, in the full domain, it is incident to an element in Ω_E, and where the definition of *borderVertex* is given in Section 3.3. The function *incidentElts* denotes the elements incident to node n in mesh m.

For instance, if the mesh in Fig. 9 represents a full domain Ω, and the elements t_0 and t_1 represent a subdomain Ω_I, then nodes v_2 and v_3 in the subdomain satisfy *gamma*, since they are on the subdomain border and, in the full domain, are incident to t_2, an element in Ω_E.

Making use of *gamma*, we can now consider ways of handling the nodes on Γ to produce an effective boundary condition. To begin thinking about how that might work, we could imagine something simple: setting the final wet–dry states of interface nodes in a subdomain run with the final wet–dry states from their full domain counterparts, but this only pushes our problem back a level: *intermediate* wet–dry states of nodes on Γ during execution of the wet–dry algorithm are unaccounted for, and these intermediate states are needed to determine the final states of nodes *just inside* Γ. In this case, *SameFinalStates* produces counterexamples, as expected, with the smallest having just two elements in Ω and one in Ω_I.

An alternative might be to initialize the wet–dry states of subdomain nodes on Γ with final wet–dry states from the full domain. In other words, the initialization step of *SameFinalStates* might instead be defined as follows:

```
all n: Sub.nodes | n.W.(s[0]) = (gamma[Sub, n] implies n.W.(f[5]) else n.W.(f[0]))
```

where the initial wet–dry states of subdomain nodes on Γ, $n.W.(s[0])$, are set to the final states of their full domain counterparts, $n.W.(f[5])$. This approach also fails, again producing a counterexample with just two elements in Ω.

As one might suspect, additional care must be taken to account for intermediate wet–dry states. In each part of the wet–dry algorithm, then, we might make use of those intermediate states from a full run. Looking at just part 2 of the wet–dry algorithm, for instance, the following predicate treats nodes on *gamma* in a subdomain run specially:

```
pred part2 [m: Mesh, s, s': State] {
    noElementChange[m, s, s']
    all n: m.nodes {
        n.W.s' = n.W.s
        n.Wt.s' = (gamma[m, n] implies ⟨state from full run⟩                    -- impose boundary condition
                        else (make_wet[m, n, s] implies True else n.Wt.s))
    }
}
```

where the expression in angle brackets ($\langle\rangle$) is to be replaced by a term representing a wet–dry state from a full run, such as $n.Wt.(f[2])$, the temporary wet–dry value in state 2 from the full run, which could be passed into part 2 from *SameFinalStates* and *solve*. Each individual part of the algorithm, then, could make use of its own intermediate state from the full run, e.g., $f[0]$, $f[1]$, $f[2]$, ..., as needed.

This more conservative approach, where we pull out every intermediate wet–dry state along Γ from a full run, indeed results in a correct implementation, and does so without the need of any element state. Through a process of experimentation, however, we find we can get by with fewer states from a full run.

Previously we noted that failure to enforce boundary conditions leads to counterexamples that fall into two broad categories. In the full run, these behaviors correspond to computations performed in part 2 (*propagate wetting across an element*) and part 4 (*make landlocked nodes dry*) of the wet–dry algorithm, and it is in just these parts that boundary conditions must be applied. When the predicates for parts 2 and 4 are modified as above, we are able to show that using just the *final* wet–dry states from a full run (i.e., $n.W.fo/last$) in those parts is sufficient to satisfy the *SameFinalStates* assertion. So, while we could store and make use of every intermediate wet–dry state of every boundary node, doing so is unnecessary.

Thus, for actual simulations in ADCIRC, we can record a minimal amount of data from a full run—the final wet–dry states of nodes on Γ—and during a subdomain run, force those states in parts 2 and 4 of the wet–dry algorithm. This is in fact how subdomain modeling is implemented in ADCIRC beginning with v51.42 [29].

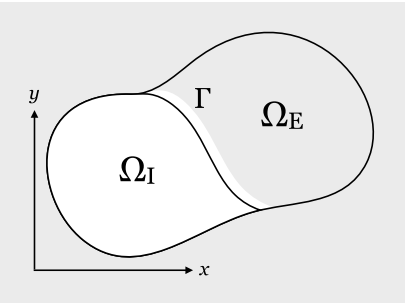


Fig. 13. Mesh for full run.

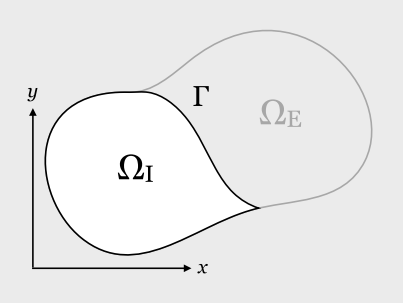


Fig. 14. Mesh for subdomain run.

To perform the analysis above, Alloy offers a variety of solvers, including MiniSat [35], an incremental SAT solver from Chalmers University of Technology, Sweden, and Lingeling [9] from Johannes Kepler University, Austria. Checking the *SameFinalStates* assertion on the same 2.8-GHz-Intel-Core-i7 laptop computer with the above changes, for as many as 4 elements and 6 nodes requires a little over 3 minutes with MiniSat, but just 30 seconds with Lingeling. If we add as a fact the *NoCutPoints* assertion previously checked in Section 3.3, that time further drops to about 18 seconds. For as many as 5 elements and 7 nodes, Lingeling requires about 13 minutes. In all cases, Alloy finds no counterexamples to the assertion.

Though such an analysis is bounded, the rationale for undertaking it springs from an appeal to the *small scope hypothesis:* if an assertion is invalid, it probably has a small counterexample [26], as we have seen in preceding examples. That those encountered have been small is also consistent with a property required for the solution of hyperbolic PDEs, the so-called Courant–Friedrichs–Lewy (CFL) condition [28], which suggests a limited range of data dependencies in a single time step.

5.2. An alternative approach for full and subdomain runs

To accommodate full and subdomain runs, other approaches can be taken, and we sketch one such alternative here that employs a different strategy. By once again making use of predicate abstraction, we can have a single mesh instance do double duty and serve the needs of both.

To do so, we consider a mesh that represents just region Ω_I, and give it a different interpretation in each case. For a full run, nondeterminism is added along Γ to model arbitrary behavior external to it, as depicted in Fig. 13 (via the narrow white band extending beyond Γ into Ω_E). For a subdomain run, we use that same region as is, without the added nondeterminism, as shown in Fig. 14 (in white).

As a first step, we define *Interface* nodes that may appear only on the border of a mesh:

```
sig Interface extends Node {
    allowsWetting, preventsDrying: Bool
}
fact { all m: Mesh, i: Interface | borderVertex[m, i] }
```

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.preventsDrying* is likewise true when such an element is active.

The conditions defined by the two fields can then be used to affect how nodes become wet or dry to account for interactions with Ω_E in a full run. The predicate *make_wet*, for instance, would then be modified as follows:

```
pred make_wet [m: Mesh, n: Node, s: State] {
    (some e: m.elements | e.slowFlow = False and loneDryNode[n, e, s])
    or (s in F and n in Interface and n.allowsWetting = True)        -- allow wetting in a full run
}
```

so that an imaginary element in region Ω_E can possibly *allow* a node to become wet, where F is an extension of *State* for full runs, as before. With a similar change in *make_dry*, an imaginary element in region Ω_E can *prevent* a node from becoming dry. Intuitively, the updated predicates make clear that subdomains require state from a prior full run on Γ if they are to produce final wet states that are equivalent to their full domain counterparts.

If we enforce boundary conditions as before in parts 2 and 4 of the wet–dry algorithm, we are again able to show that the *SameFinalStates* assertion is satisfied, and is so for larger mesh instances, in this case for as many as 8 elements and 10 nodes in about 3 minutes.

6. Role in understanding empirical algorithms

Based on empirical rules, the wet–dry algorithm has behaviors that may not be obvious from inspection alone. Indeed, the algorithm emerged piecemeal over time from the experience and analysis of users and developers, not from a specification. While some behaviors may appear in the course of using ADCIRC, understanding the implications of empirical algorithms may be better achieved through complementary means. By specifying properties of interest in Alloy, for instance, we are able to probe those behaviors. Here we give an example.

Our look at model checking began with a question developers raised but were unable to answer without resorting to experiments: *why would an element with three wet nodes be dry?*

ADCIRC allows users to define virtual "recording stations," whose geographic locations lie somewhere in a finite element mesh and whose physical values, such as water surface elevation, are determined by interpolating within an element from nodal quantities. When the associated element is dry, ADCIRC produces no output—by design—for the station even if its nodes are wet. Users noticed the apparent discrepancy between the wet–dry states of elements and nodes, and wondered how an element with three wet nodes could be dry.

6.1. Nature of the approximation

To answer the question using Alloy, we first take a step back and consider the nature of the approximation being made. Our model of the wet–dry algorithm can be characterized as an over-approximation since it represents more instances than are actually present. Because checking the subdomain modeling extension involves a safety property, however, its conclusions are meaningful: if a property is true in the model it is also true in the concrete system. If, on the other hand, we are seeking an instance that satisfies a property of interest, such as a witness to the question above, then an over-approximation may lead to spurious results.

In our model, we rely on predicate abstraction, and while doing so we largely avoid over-approximation. For instance, part 2 of the algorithm uses the term *e.slowFlow*, which is defined to be true when $V_{ss}(e) \leq V_{min}$ for an element e. This abstraction is exact in the sense that the actual steady-state velocity, V_{ss}, can take on arbitrary values that are both physically realizable and independent of other tests in the algorithm. The same argument can be made for the water column height, H_n, as it appears in parts 1 and 3 of the algorithm.

On the other hand, the term *e.lowNode* in part 3 is the element's node with the lowest water surface elevation, and at present, an inconsistency can arise between adjacent elements in the way the nodes are selected. To prohibit physically impossible instances from being generated, we insist on a total ordering for nodes, as though by water surface elevation, and use that in selecting the lowest node in each element.

We begin by adding a field *lte* to mesh extension *M* and then require that it be a total order over the set *M.nodes*:

```
one sig M extends Mesh { lte: Node → Node }
fact { totalOrder[M.lte, M.nodes] }
```

Then, an element's *lowNode* is defined to be the minimum of its incident nodes, ensuring that any witness generated is one that is physically meaningful:

```
fact { all e: M.elements | e.lowNode = min[M.lte, dom[e.edges]] }
fun min [r: univ → univ, s: set univ ]: univ {
   { y: s | (no x: s | x → y in r − iden) }
}
```

where, using total order r, the function *min* returns the unique minimum over a (non-empty) set s, and where *iden* is the identity relation.

We remark that, while the water surface elevations of two or more nodes in ADCIRC could in fact be equivalent, the formulation above suffers no loss of generality. What it prevents is the following: a pair of adjacent elements e_1 and e_2 with edges (e_1, u, v) and (e_2, v, u), where $e_1.lowNode = u$ and $e_2.lowNode = v$. In other words, ADCIRC must select a node even in the case of a tie, and we continue to leave that choice undetermined[2] so long as adjacent elements avoid disagreeing about which of their common nodes has the lowest water surface elevation.

6.2. Generating witnesses

We now perform a simulation and look for an instance satisfying the property of interest, a dry element with three wet nodes:

[2] Since arbitrary total orders are allowed.

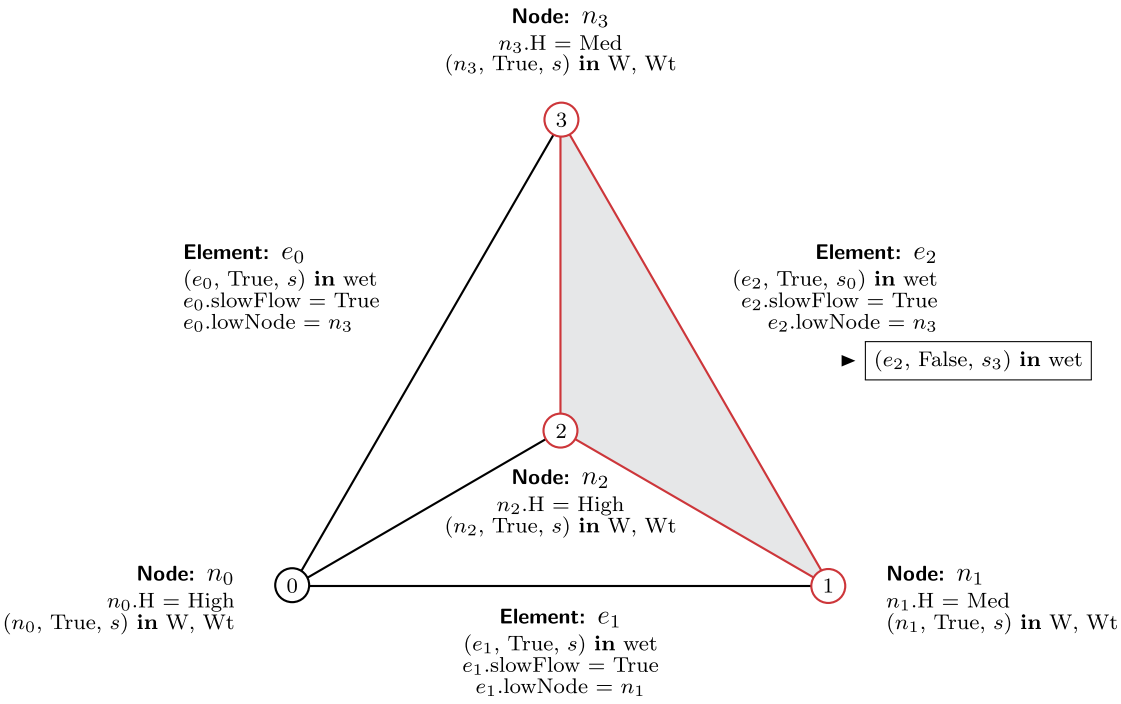


Fig. 15. Dry element e_2 with wet nodes n_1, n_2, and n_3, where the state change in e_2 is boxed.

```
pred dryElementWetNodes {
    let s = toSeq[so/first, so/next] | solve[M, s]
    some e: M.elements | e.wet.so/last = False and wetNodes[e, so/last] = 3
}
```

where *e.wet.so/last* denotes the final wet state of element *e*, and where the definition of *wetNodes* is given in Section 4.3. We start by specifying a small scope in the Alloy analyzer and gradually increase it until an instance is found with 3 elements and 4 nodes, the smallest such instance in terms of elements and nodes. By making use of the incremental solving capability of MiniSat, we can generate and step through instances within that scope, one at a time.

Generally speaking, because of the sheer number of satisfying combinations of states and possible traces, there exist many such ways of producing a dry element with three wet nodes, even within this limited scope. For a simpler instance, we can ask for one with the fewest number of wet–dry state changes by initializing nodes to be wet. One such example is shown in Fig. 15, which is annotated with the states of both nodes and elements, and where a single state change is boxed, in this case for element e_2, showing that it dries in state s_3. For brevity, where s appears in fields that vary with state (W, Wt, and *wet*), it denotes the set of all states, i.e., $s = \{(s_0), (s_1), (s_2), (s_3), (s_4), (s_5)\}$.

Recall in the wet–dry algorithm that elements always start out wet but become dry if either of their nodes with the highest water surface elevations has a water column height less than $1.2H_{min}$, a user-defined constant, or is not *High*, as it appears in our Alloy model. Then, since $e_2.lowNode = n_3$, that leaves n_1 and n_2 as the nodes with the highest water surface elevations, and $n_1.H$ is *Med*, not *High*, so it is clear why element e_2 dries: it is just part 3 of the wet–dry algorithm restated. Perhaps, then, a more appropriate framing of the question would be to ask what prevents wet nodes on a dry element from drying, since we already know why nodes might become wet and why elements might become dry.

Continuing with the instance shown in Fig. 15, after element e_2 dries in part 3, the wet–dry algorithm checks for nodes that are landlocked in part 4, making dry any that are found. Recall that a node is landlocked if and only if it is incident only to inactive elements, and an element is active if and only if it is wet and has three temporarily wet nodes. So the linchpins in the instance shown are really nodes n_0 and n_2: their water column heights H are *High*, which simultaneously prevents elements e_0 and e_1 from drying and thereby taking down every incident node with them by making them dry, since they would be landlocked: there would be no active elements to keep nodes from drying. What is clear is that nearby nodes with high water column heights will be found whenever a dry element has three wet nodes.

For studies exploring the physical basis of the wet–dry algorithm's empirical rules, we imagine that the ability to answer what-if questions in this manner could be helpful, since they can be readily posed, and because witnesses can often be found instantaneously, as they are above. Instead of relying solely on a finite element solver, then, and hoping for a satisfying instance or counterexample by chance, we can determine whether such a situation is possible, within bounds, and provide a small witness that confirms it.

6.3. Ongoing debate

Using ADCIRC, another developer answered the question by instrumenting the code and performing a simulation to see if such an instance could be found. By simulating tidal hydrodynamics around Shinnecock Inlet on Long Island, New York, with a mesh of 5 780 elements and 3 070 nodes—and about 5 CPU-hours of run time—he also concluded that "an element can be dry when all three nodes around the element are wet." The exercise led to other questions about whether flow is "shut down" in places where this occurs, and whether such situations might degrade numerical stability while offering little improvement in wet–dry accuracy.

What this points to is an ongoing debate about the current scheme, which was put into place in order to reduce mass balance errors due to flow down steep slopes. It has been noted that, while improving the physics in some cases, it can also artificially elevate water levels in marshes and tidal creeks during tidal simulations, so a resolution does not appear to be immediately forthcoming. But opinions are varied, and when another developer learned about our model checking approach, he asked whether we agreed with his conclusion that "if all three nodes are wet, an element *should* be wet."

While not taking a position on the issue, we believe that, in light of what are likely to be ongoing challenges, a decision support system based on model checking could be helpful in exploring possible realizations of wet–dry algorithms. To simplify their expression, a domain-specific description language could be designed, perhaps drawing on an existing, imperative extension to Alloy [32]. Instances and counterexamples might then be depicted in a more intuitive way for ocean modelers, as in Fig. 15, using the VLM tool for domain-specific visualizations [17], Alloy's API in Java, or αRby [31], an embedding of Alloy in Ruby.

7. Related work

While control systems, communication protocols, and hardware design are historically among the most common applications of formal methods [11,40], several studies do address problems related to scientific computing.

Siegel et al. [34] present a framework that tests small numerical programs for *real equivalence*, meaning that one program can be transformed into the other using the identities of real numbers. The approach is based on creating a sequential program that serves as a specification, and then using it as the measure against which an implementation is compared, such as a more complex MPI-based parallel program. Equivalence checking is performed by building up symbolic expressions in both programs and comparing them using the SPIN model checker. Instead of working with models of programs, then, the idea is to work directly with programs themselves.

Arnold et al. [3] address the complexity of reasoning about imperative implementations of sparse matrix formats, an essential element of large-scale problems in scientific computing. The authors define an APL-like functional programming language that allows a more natural expression of operations on such formats. An associated verification approach translates programs written in the language into Isabelle/HOL using a framework that is able to automate the proofs in many cases.

Chadha et al. [10] describe a methodology for developing concurrent systems that extends the Larch family of specification languages with the CCS process algebra. While not designed especially for numerical algorithms, the approach is applied to the specification of a parallel preconditioned conjugate gradient solver for linear systems of equations, the type of solver often used in finite element analysis systems like ADCIRC.

In other studies, Bientinesi et al. [8] use Floyd–Hoare logic to derive and prove the correctness of dense linear algebra algorithms. Baugh [5] formalizes portions of a finite element analysis library using algebraic specifications in the style of the Larch Shared Language.

8. Conclusion and future work

As far as we are aware, our study is the first to look at formal methods like Alloy in the context of scientific computing and large-scale physical simulation. By supporting experimentation with implementation choices and providing feedback, the approach helps increase our confidence that the boundary conditions in our subdomain modeling extension are implemented correctly.

We have already noted the model-finding capabilities of Alloy that support the generation of arbitrary mesh topologies. Among its other attractive features is support for the incremental construction and analysis of models, which we employed both in development and in our presentation; such an approach is facilitated by a combination of a declarative style of specification and its rich data modeling features, including subtyping. It is our perception that these aspects of Alloy enabled us to create and discover good abstractions and invariants more easily than would have been possible using a programming language, and that such products are precisely what are needed by a correspondingly good implementation. And it is this experience that leads us to conclude that Alloy might serve as a good "everyday" tool for those working in traditional areas of science and engineering, particularly since it supports the analysis of sometimes trickier issues if and when they arise.

Whether and how specification and verification approaches might eventually be applied more broadly to the development of scientific software are open questions. Some of the techniques that automatically extract models from code—SLAM [4], Java Pathfinder [39], and Siegel's related approach with its emphasis on floating-point expressions [34]—may make model checking more accessible to a wider community. One can easily imagine scenarios for functional verification

requiring symbolic, real-valued expressions, in the style of Siegel, or matrix operations, though we wonder if using conventional programming languages is the ideal means for doing so.

Our approach, by way of contrast, involves us in the development of *models* of software, which may offer some advantages: scientists and engineers are accustomed to working with models anyway, and with automatic, push-button analysis as an alternative to theorem proving, one can focus on modeling and design aspects. As Jackson writes [26], code is a poor medium for exploring abstractions. It is not clear to us in any case how one would go about tackling the analysis we perform here using, say, Fortran code, or whether any insights can be gained by doing so.

If tools like Alloy are to find practical application in scientific and engineering domains, one might hope for additional support for numerical expressions, thereby expanding the scope of analyses that can be performed. A modest step might be incorporating an SMT solver like Yices in Alloy [15], or perhaps dReal [18], which is capable of handling first-order logic formulas over the reals. But more than any extension to Alloy, what would have benefited our study most is a tool capable of automatically producing planar embeddings of meshes from Alloy instances, which proved to be tedious to do by hand.

Despite our positive experiences working in the given domain of application, we must acknowledge that scientific software is often developed from a different perspective, where global, non-reentrant data structures are common and short-term concerns about performance dominate. Indeed, to realize a new, adaptive approach to subdomain modeling—not described here—we found ourselves reimplementing ADCIRC almost entirely in a prototype developed with abstraction in mind [2]. Our future directions are focused on that and similar efforts that incorporate adaptivity both for re-analysis and mesh refinement. We expect to make use of Alloy's support for experimenting with abstractions, building object models, and finding representation invariants [24].

Acknowledgements

The authors are grateful to the reviewers of Science of Computer Programming and ABZ 2016 for their careful reading and suggestions for improvement. Thanks are also due to the Department of Civil, Construction, and Environmental Engineering at NC State University for providing an environment that encourages interdisciplinary research.

References

[1] Alloy models from the paper, http://www4.ncsu.edu/~jwb/alloy/.
[2] A. Altuntas, J. Baugh, Adaptive subdomain modeling: a multi-analysis technique for ocean circulation models, Ocean Model. 115 (2017) 86–104.
[3] G. Arnold, J. Hölzl, A.S. Köksal, R. Bodík, M. Sagiv, Specifying and verifying sparse matrix codes with Isabelle/HOL, ACM SIGPLAN Not. 45 (2010) 249–260.
[4] T. Ball, S.K. Rajamani, The SLAM toolkit, in: International Conference on Computer Aided Verification, Springer, 2001, pp. 260–264.
[5] J.W. Baugh Jr., Using formal methods to specify the functional properties of engineering software, Comput. Struct. 45 (1992) 557–570.
[6] J. Baugh, A. Altuntas, Modeling a discrete wet–dry algorithm for hurricane storm surge in alloy, in: International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, Springer, 2016, pp. 256–261.
[7] J. Baugh, A. Altuntas, T. Dyer, J. Simon, An exact reanalysis technique for storm surge and tides in a geographic region of interest, Coast. Eng. 97 (2015) 60–77.
[8] P. Bientinesi, J.A. Gunnels, M.E. Myers, E.S. Quintana-Orti, R.A. van de Geijn, The science of deriving dense linear algebra algorithms, ACM Trans. Math. Softw. 31 (1) (2005) 1–26.
[9] A. Biere, Lingeling, Plingeling and Treengeling entering the SAT competition 2013, in: Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions, in: Department of Computer Science Series of Publications B, vol. 2013-1, University of Helsinki, 2013.
[10] H.S. Chadha, J.W. Baugh Jr., J.M. Wing, Formal specification of concurrent systems, Adv. Eng. Softw. 30 (1999) 211–224.
[11] E.M. Clarke, J.M. Wing, Formal methods: state of the art and future directions, ACM Comput. Surv. 28 (1996) 626–643.
[12] J.C. Dietrich, R.L. Kolar, R.A. Luettich, Assessment of ADCIRC's wetting and drying algorithm, Dev. Water Sci. 55 (2004) 1767–1778.
[13] J.C. Dietrich, R.L. Kolar, J.J. Westerink, Refinements in continuous Galerkin wetting and drying algorithms, in: Proc. Ninth Int. Conf. on Estuarine and Coastal Modeling, 2006, pp. 637–656.
[14] T. Dyer, J. Baugh, SMT: an interface for localized storm surge modeling, Adv. Eng. Softw. 92 (2016) 27–39.
[15] A.A. El Ghazi, M. Taghdiri, Analyzing Alloy constraints using an SMT solver: a case study, in: 5th International Workshop on Automated Formal Methods (AFM), 2010.
[16] FEMA, Flood Insurance Study: Southeastern Parishes, Louisiana, Intermediate Submission 2: Offshore Water Levels and Waves, U.S. Army Corps of Engineers, New Orleans District, 2008.
[17] L. Gammaitoni, P. Kelsen, Domain-specific visualization of Alloy instances, in: International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, Springer, 2014, pp. 324–327.
[18] S. Gao, S. Kong, E.M. Clarke, dReal: an SMT solver for nonlinear theories over the reals, in: International Conference on Automated Deduction, Springer, 2013, pp. 208–214.
[19] S. Graf, H. Saïdi, Construction of abstract state graphs with PVS, in: Proceedings of CAV '97, in: Lect. Notes Comput. Sci., vol. 1254, Springer, 1997, pp. 72–83.
[20] L. Graham, Adaptive Measure-Theoretic Parameter Estimation for Coastal Ocean Modeling, Ph.D. thesis, The University of Texas at Austin, 2015.
[21] L. Graham, T. Butler, S. Walsh, C. Dawson, J.J. Westerink, A measure-theoretic algorithm for estimating bottom friction in a coastal inlet: case study of Bay St. Louis during Hurricane Gustav (2008), Mon. Weather Rev. 145 (3) (2017) 929–954.
[22] J. Haddad, S. Lawler, C.M. Ferreira, Wetlands as a nature-based coastal defense: a numerical modeling and field data integration approach to quantify storm surge attenuation for the Mid-Atlantic region, in: OCEANS 2015, MTS/IEEE, Washington, 2015, pp. 1–6.
[23] J.L. Hanson, H.M. Wadman, B. Blanton, H. Roberts, Coastal Storm Surge Analysis: Modeling System Validation, Report 4: Intermediate Submission No. 2.0 (No. ERDC/CHL-TR-11-1), U.S. Army Engineer Research and Development Center, Kitty Hawk, NC, 2013.
[24] D. Jackson, Object models as heap invariants, in: Programming Methodology, Springer, 2003, pp. 247–268.
[25] D. Jackson, Software Abstractions: Logic, Language, and Analysis, revised edition, MIT Press, 2012.
[26] D. Jackson, C.A. Damon, Elements of style: analyzing a software design feature with a counterexample detector, IEEE Trans. Softw. Eng. 22 (1996) 484–495.

[27] D. Jackson, J. Wing, Lightweight formal methods, Computer 29 (1996) 22–23.
[28] R.J. LeVeque, Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems, SIAM, 2007.
[29] R.A. Luettich, J.J. Westerink, Formulation and numerical implementation of the 2D/3D ADCIRC finite element model, version 44.xx, http://www.adcirc.org.
[30] S.C. Medeiros, S.C. Hagen, Review of wetting and drying algorithms for numerical tidal flow models, Int. J. Numer. Methods Fluids 71 (2013) 473–487.
[31] A. Milicevic, I. Efrati, D. Jackson, αRby—an embedding of Alloy in Ruby, in: International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, Springer, 2014, pp. 56–71.
[32] J.P. Near, D. Jackson, An imperative extension to Alloy, in: International Conference on Abstract State Machines, Alloy, B and Z, Springer, 2010, pp. 118–131.
[33] Y.P. Sheng, J.R. Davis, R. Figueiredo, B. Liu, H. Liu, R. Luettich, V.A. Paramygin, R. Weaver, R. Weisberg, L. Xie, L. Zheng, A regional testbed for storm surge and coastal inundation models—an overview, in: Proceedings of the 12th International Conference on Estuarine and Coastal Modeling, St. Augustine, FL, ASCE, 2012, pp. 476–495.
[34] S. Siegel, A. Mironova, G. Avrunin, L. Clarke, Combining symbolic execution with model checking to verify parallel numerical programs, ACM Trans. Softw. Eng. Methodol. 17 (2008), Article 10.
[35] N. Sorensson, N. Een, MiniSat v1.13 – a SAT solver with conflict-clause minimization, in: SAT 2005 Competition, 2005.
[36] U.S. Army Corps of Engineers, Performance Evaluation of the New Orleans and Southeast Louisiana Hurricane Protection System, Volume I—Executive Summary and Overview. Final Report of the Interagency Performance Evaluation Task Force, U.S. Army Corps of Engineers, Washington, DC, June 2009.
[37] U.S. Army Corps of Engineers, Elevations for Design of Hurricane Protection Levees and Structures, Lake Pontchartrain and Vicinity, West Bank and Vicinity, New Orleans to Venice, Louisiana Project. Report Version 2.0, U.S. Army Corps of Engineers, New Orleans District, December 2014.
[38] M. Van Ledden, M. Kluyver, A.J. Lansen, R. Kluskens, Highlights of Dutch and US coastal graduation projects in the Mississippi Delta after Hurricane Katrina, in: Jubilee Conference Proceedings, NCK-Days 2012: Crossing Borders in Coastal Research, Enschede, Netherlands, 13–16 March 2012, University of Twente, 2012.
[39] W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda, Model checking programs, Autom. Softw. Eng. 10 (2003) 203–232.
[40] J. Woodcock, P.G. Larsen, J. Bicarregui, J. Fitzgerald, Formal methods: practice and experience, ACM Comput. Surv. 41 (2009) 19:1–19:36.
[41] O.C. Zienkiewicz, R.L. Taylor, P. Nithiarasu, The Finite Element Method for Fluid Dynamics, 7th edition, Butterworth–Heinemann, Oxford, UK, 2013.