Technical guidance for the inclusion of models/modules in the NextGen Water Resources Modeling Framework

Fred Ogden, Nels Frazier, Keith Jennings, Jonathan Frame, Wouter Knoben, Tadd Bindas, Yalan Song, Irene Garousi-Nejad, Jeffrey Carver, Andy Wood, Anthony Castronova, Arpita Patel, Shahabul Alam, Sifan A. Koriche, Junwei Guo, Cyril Thébault, Raymond J. Spiteri, Ahmad J. Khattak, James Halgren, Patrick J. Clemins, Mukesh Kumar, and Martyn Clark

1. Introduction

NOAA/OWP started development of the "Next Generation Water Resources Modeling Framework" (nicknamed "NextGen") to provide flexible options to experiment with representations of the hydrologic cycle in a model-agnostic framework. The NextGen Framework is a model-agnostic, standards-based, model interoperability software tool that allows explicit coupling of models through the sequential sharing of computed states and/or fluxes between different domain science models or modules. The intent is that NextGen empowers the broader science and applications community to collaborate on water resources modeling problems.

Identification of general design requirements for the NextGen Framework occurred during Interagency meetings held 26-28 October, 2020, involving NOAA, USGS, USACE, USBR, and DOE. These design requirements resulted from discussions around the question "What features would a useful modeling framework possess?" Resulting design requirements included the following:

- Maximum flexibility - as models, data sources, and needs evolve, the framework supports changes and additions
- Model agnostic
- Common architecture to avoid duplication and promote interoperability
- Open source development
    - Promote code reuse and development efficiency
    - Authoritative repository for federal water models
    - Ease/encourage participation by partners and community
- Apply standards where applicable and necessary
    - Coding, coupling
    - Data and metadata
    - Model verification/validation and test data
- Friendly to domain scientists and engineers to facilitate community development
    - Avoid wholesale rewriting of domain science code
    - Encourage and ease adoption
- Sharing models, data, and results
    - Library of model codes and data sets
    - Evaluation tools
- Establish and maintain a glossary and define terms to communicate clearly across disciplinary boundaries
- Use mature open source libraries where appropriate
- Multi-language support (C++, C, Fortran, Python)

- Run on hardware from laptops to supercomputers
- Two-week target to allow graduate students or new employees to add functionality
    - Excellent documentation and step-by-step examples/tutorials
    - Programming required but not computer science background

A second interagency meeting held in January, 2021, sought to identify existing standards to both couple hydrologic and hydraulic models as well as describe the hydrologic and hydraulic features of the landscape. The chosen standards include (a) the CSDMS Basic Model Interface (BMI), a model coupling standard; and (b) the Open Geospatial Consortium WaterML, version 2.0, part 3, Hydrologic Features (HY_Features) conceptual data model.

A key need for ongoing NextGen development is to promote consistency in the application of the BMI standard in a way that provides maximum flexibility and information required for its successful use within the NextGen Framework. **To address this need, this document defines technical standards, considerations, and suggestions for future requirements to support the development of the NextGen framework.** The document also provides suggestions for the development and application of models that are intended to be coupled within the framework. The guidance provided in this document is intended to improve the functionality of coupled NextGen instantiations.

The remainder of this document is organized as follows. Section 2 defines the assumptions that guide efforts in the development of the NextGen framework. Section 3 defines the strategy and best practices for NextGen code development, including the BMI implementation, the initialize-update-finalize paradigm, model testing, and best practices for community modeling. Section 4 summarizes the metadata to integrate models/modules in NextGen, including definition of model variables, time information, and calibration parameters. Section 5 defines the information that is useful to expose in NextGen applications, including model setup workflows, information on forcing data, and restart capabilities. Finally, Section 6 defines key needs for further research and development, and Section 7 provides a summary of key NextGen contributions.

2. Guiding assumptions

With the general design requirements in mind and with respect to the intent of this document to guide efforts in the development of the NextGen framework and in encouraging community contributions to the NextGen ecosystem:

- A goal of developing the NextGen framework is to accelerate the rate of advances in hydrologic science and its application to water prediction and forecasting.
- NextGen is designed to provide a flexible, interoperable, open-source, and standards-based framework to accelerate experiments in support of that goal.
- As development continues on the NextGen framework itself and as models are brought within the framework, we are providing these recommendations, based on experience, to promote effective collaborative development towards our goal.

3. Strategy and best practices for NextGen code development

The diversity of choice in modeling approaches in NextGen gives rise to many choices a developer should consider when readying their model or module for use in NextGen. In this

section, we define the strategy for NextGen development and code development best practices to promote consistency in the application of the BMI standard.

3.1 BMI implementation

BMI, created by Community Surface Dynamics Modeling System (CSDMS) at the University of Colorado, Boulder, provides an interface specification that lets the NextGen framework control model runtime and pass data from one model to another.

BMI specifies a set of functions that can be broken unto the following categories:

- Model information functions
- Variable information functions
- Time functions
- Model control functions
- Variable getter and setter functions
- Model grid functions

A full discussion of every function within each category is outside of the scope of this document, but we describe the functions critical for NextGen in Appendix A. Readers may consult the BMI wiki from CSDMS for more information.

We note here that every BMI function must be implemented in order for a model to run in NextGen. In many cases, this requires that the necessary functionality or data be accessible via BMI, which may or may not require code refactoring. However, there are some cases where a BMI function is not applicable and can be set to return “BMI_FAILURE” (e.g., a model with a lumped discretization does not need any of the model grid functions that describe spatially distributed configurations). There is, however, a minimum set of BMI functions that the NextGen model engine expects to be implemented (Appendix A).

NextGen is a model- and language-agnostic framework, meaning it is compatible with various hydrologic models written in different programming languages. The framework currently supports models written in the following programming languages:

- C
- C++
- Fortran
- Python

The NextGen model engine requires one or more additional non-BMI function implementations for C, C++, and Fortran modules. These functions are necessary for the model engine to dynamically load the module into the simulation runtime environment. The registration functions are summarized in Appendix B.

3.2 Initialize-update-finalize paradigm

Execution of models within the NextGen framework requires that models employ a initialize-update-finalize paradigm that underlies the Basic Model Interface (BMI, explained in more detail in the following section). Adapting existing model codes to the BMI standard requires access to model functionality while allowing the framework to control model startup,

execution through time, and termination. This means the model must include clearly defined routines for the following three functions:

- **Initialize**. Initialize performs important operations, including the creation of the model instance and related objects, the opening and reading of necessary files, and the configuration of model options and parameters. To the last point, we recommend that all models employ configuration files where the user can specify option flags and parameters values. These should be set at runtime, not at compile time.
- **Update.** Update executes the model for a single time step. This action should be self-contained and not include any initialization code. Additionally, the developer should ensure that no update code is executed in the model driver or main program. If this occurs, then the update function is not self-contained, and the related BMI functions will not return expected behavior.
- **Finalize**. Finalize is the final step of a model run, and it includes the code to close open file connections and clean up memory.

Note that the initialize-update-finalize paradigm can be implemented at multiple levels of process granularity, including an entire hydrologic model, model sub-domains (e.g., vegetation, snow, soil, aquifer), model integrators (e.g., the temporal evolution of a state variables in a given sub-domain), and flux calculators (specific fluxes for a given state variable through a given model element control surface). In addition to execution using the BMI, models can include an option to run in stand-alone mode outside the framework, a useful feature for development and testing. Alternatively, developers can write their own calling program using the BMI to test their model.

3.3 Model testing

3.3.1 Unit Testing

Unit testing should be used with all modules (Kolawa and Dorota, 2007). By design, this requires testable units of code. These units should naturally align with the computational model being considered. Each unit of code should have associated tests, written independently from the module library code, which test the units under various conditions, e.g. with known input values, bad input values, incompatible arguments/parameters, etc. Each test should know what the expected result of calling the unit of code should be, and the test should assert the intended behavior/result is achieved.

These tests should exercise the code to ensure proper implementation of code as well as numerical models. Variable initialization, if/else logic on boolean function parameters, data type conversion/assumptions, ect should be tested in these unit tests. In a well designed module, these tests can extend into stand-alone tests of numerical modeling criteria,

This requires defining functional unit tests, e.g., testing to verify conservation and diagnostic variables meet the criteria from known starting conditions and inputs match known solutions, etc.

3.3.2 Enforcing relevant conservation laws

If the model component is a “state integrator” then the component should expose the fluxes at the boundaries of the control volume that the component represents -- along with the change in state variables over the time interval that are computed using the BMI functions `update()` or `update_until()`. This will require flux-to-state mapping (i.e., the states that a flux depends on) in order to determine the fluxes that affect the time evolution of model states.

One suggestion is that the modeler expose a variable through the BMI interface named bmi_conservation_array, that is dimensioned by conservation quantity (e.g. mass, momentum, energy), and by control volume number or index. Ideally the value of each element of this array shall be near zero (or other target value specified by the model developer), indicating that the model/module is conserving that quantity in that control volume.

3.3.3 Quantifying numerical errors

This will need to be done at multiple levels of process granularity -- e.g., for individual model components (e.g., temporal truncation errors) and for system-scale predictions (e.g., splitting errors).

3.3.4 Detecting misbehaving modules

Runtime Conservation Law Checks

Conservation law checks allow detection of misbehaving modules. This requires that the model definition file includes information that tells the framework which fluxes (F) and state variables (S) are used to calculate conservation, and the form of the equation used, for example:

$S(m,t) + (F(i,j_1) + F(i,j_2) + F(i,j_3))*gdt - S(m,t+1) < target$	(1)

where:

$S(m,t)$ = State variable name in control volume m variable at time t

$F(i,j)$ = Flux i at control surface j_k

gdt = integrating time step

$target$ = the tolerance value that defines acceptable convergence

The framework will detect a problem if the application of equation (1) across all relevant fluxes and storages fails in any instance to produce a result that causes the inequality to be false. Implementation of this check will require an entry in the model definition file giving the names of the storage and flux variables, maximum number of storage control volumes and n-tuples of flux control surface indexes for each control surface. This method will work best for simpler

modules. By definition, a positive flux is a flux leaving the control volume. Negative numbers in the index below indicate that the relevant flux is defined relative to a neighboring control volume.

Runtime Internal Stability Checks

A second means to detect misbehaving models/modules relies on internal stability or conservation law checks that expose logical variables to the BMI interface. This requires that the user provide the framework the logical variable name, specify its role, and the error value and an error message. When the model/module produces a stability check with a value that equals the error value, then the framework would know that model has detected an internal problem. This method will work best for more sophisticated models.

Example:

variable_name internal_stability_check 1 "Model *name* courant number exceeded 1.0".

3.3.5 Benchmarking module capabilities/performance

Benchmarking expands on unit tests by providing examples of expected model output behavior. This can take the form of functional relationships: plots of storage in a control volume against fluxes at the boundaries of the control volume, at any level of aggregation. This gives a straightforward overview of a model/module's capabilities for comparison against other model/module's at the same level of process granularity.

This could also include known performance determined from comparison against observation data in standard test cases, possibly including typical computation times so the user can choose their preferred trade-off between model accuracy/realism and computational cost. Sharing the evaluation benchmark settings (datasets, training/calibration/validation periods and sites) would be critical in such cases. An example for streamflow simulation could be the CAMELS data set.

3.4 Community modeling

Community contributions to NextGen are managed by a distributed version control system (GitHub) using publicly available repositories.

3.4.1 Collaborative model development (commit, testing, code review)

- Commit Messages: Use clear and concise commit messages.
- Branching Strategy: Use feature branches for new developments and bug fixes.
- Writing Test Cases: Unit tests for individual components and integration tests for combined functionality. It is convention to put code for tests in its own subdirectory.
- Coding standards: Code should be easy to understand, avoid complicated or obfuscated solutions whenever possible (see Appendix D)
- Running Test Cases: Continuous integration using pipelines to run regression tests automatically.
- Peer Review: All code contributions should be reviewed by at least one other team member.

3.4.2 Code maintenance

GitHub releases, with versioning, should be deployed when the code has new features. A code maintainer should be established for open-source code developments. Code maintainer information, including ways for contacting, should be listed in the repo's README.md file.

The method for which the developer would like contributions to be made should be established in a CONTRIBUTING.md file. If the maintainer of the repository is *not* the corresponding author, the corresponding author should be mentioned in the README.md file.

An unsupported model is a broken model. A key issue is what happens after the paper is published and the research is done, and, potentially, when the main contributor has moved to a different position. The best practice is to develop a software sustainability plan with institutional support that does not depend on individual projects and individual people. Ideally, the corresponding author should remain available for any post-publication correspondence; this requirement may become more difficult to satisfy if the corresponding author has a change in their job responsibilities (e.g., a different employer or a different role in a given organization).

3.4.3 Documentation

The best practice for open source software is to define documentation for both model developers and model users.

The documentation for model users should be provided through the use of a wiki or hosted web pages located in a docs/ folder in the repository. The information should include:

- Getting Started, providing information on the main features of the code repo (mainly for new users)
- Dependencies
- How to install the code
- Tutorials or experiments, that the author would like to document
- API Reference, providing documentation of external code functions
- Contributor's guide, informing how someone can make open-source contributions to the codebase
    - When possible, contributors should use GitHub Issues instead of email communications to track bug fixes, feature requests, upgrades, etc.

The documentation for model developers should include an overview of the model code, e.g., using doxygen (https://www.doxygen.nl/) as well as coding standards (Appendix D).

Options for how to structure docs/ include:

- Markdown files
- Notebooks containing both Markdown and Code snippets
- Hosted Docs on GitHub pages
    - Mkdocs (Material Template)
        - https://squidfunk.github.io/mkdocs-material/getting-started/
        - Publishing information:
            - https://squidfunk.github.io/mkdocs-material/publishing-your-site
    - Sphinx (PyData Template)

        - https://github.com/pydata/pydata-sphinx-theme
        - Publishing information
            - https://docs.readthedocs.io/en/stable/reference/git-integration.html

To ensure function information is properly built into the web documentation, proper docstring format should be followed. For Python modules, the following docstring specs can be ported directly into documentation:

- Google Style Docstring
    - https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html
- Numpy Style Docstring
    - https://numpydoc.readthedocs.io/en/latest/format.html

3.4.4 Open source development

A key requirement is that the models included in NextGen follow an open development process. It is desirable to have code licenses as permissive as possible. Proprietary code will not be considered as a contribution to NextGen.

3.5 Other recommendations

Most models include their own code to read in forcing data and write output data; however, the framework handles these processes when a model runs in the NextGen framework. As such, it is not necessary for a model to open file connections to forcing data and create output files during the initialize step, nor is it necessary for a model to write output data during update and finish the output file during finalize. Therefore, a developer may wish to employ compiler directives that skip over these steps when a model runs in NextGen, an approach that has been used previously for some Nextgen formulations. See an example here.

Developers adapting models/modules to execute within the NextGen Framework are encouraged to retain the ability of their model/module to run in a stand-alone mode, which can greatly benefit testing/development purposes. This might be best accomplished using a BMI-capable driver program.

4. Metadata to integrate models/modules in NextGen

The combination of forcing data requirements, time implementations, spatial discretizations, hydrologic processes represented, parameterizations, and other conceptualizations produces an immense variety of hydrologic models that can be constructed using the NextGen framework. In this section, we discuss some of the most important considerations and modeling best practices that will facilitate the integration process.

4.1 Model variables (Framework exposed)

We suggest a metadata-based approach for standardized communication of model variables, time information, processes, etc. The goal of specifying metadata requirements is to ensure that developers provide all the model/module metadata that the framework needs to correctly couple models/modules and estimate parameters using community standards, such as Schema.Org and ESIP Science on Schema.Org. The representation of the following

metadata facilitates model integration, ensures code reusability, aids in model validation, supports troubleshooting, and aligns with best practices in scientific modeling.

The metadata for each model variable should include all of the information that is typically used to describe a variable in a NetCDF file as well as additional attributes about the variable type (or role). The metadata includes:

- Variable Name: This is the name specific to the model/module and may differ from the standard variable name vocabulary. Use of standard names is encouraged at the BMI interface to avoid ambiguity.
- Standard Name: The use of CSDMS standard names is a consistent and model-agnostic way to describe variable names, ensuring data can be interoperable across different models/modules.
- Long Name: This provides a description of the variable and is usually used for axis labels on plots.
- Units: Using standardized well-defined units ensures that data is interoperable and facilitates accurate unit conversions. NextGen uses the UDUNITS syntax.
- Dimension name: Defines name of the dimension. Variables can have no dimensions, in which case they are called scalars, or they can have multiple dimensions. In addition to spatial (x, y) and temporal (t) dimensions, other dimensions may be included, such as soil layers, which represent a type of depth.
- Coordinate variable. Defines the axes or coordinates of dimensions.
- Datatype: Variable type must be one of the netCDF external data types supported by netCDF-4. For example, string, char, byte, unsigned byte, short, int, float, complex, opaque, etc. Not all these types (string, char, byte, complex) are explicitly supported by BMI and/or NextGen.
- Precision Level: The number of significant digits for variables in the simulations. This can help decide the right data type for storage.
- Add offset and scale factor: These attributes are used to store data efficiently (e.g., converting double precision variables to short integers), reducing storage space while maintaining the precision.
- Method for Deriving Cell Value Data: This indicates the method used to derive data representing cell values across each dimension. For example, if a variable has a time dimension and its value is the starting value for a specified period, the temporal aggregation is period_start; if it is the ending value, it is period_end; and if it is an average over a specified period, it is period_average, which is commonly used for variables like average temperature of humidity. This property can be defined as a dictionary, for example cell_methods={"time": "period_start", "area": "mean"}.
- Spatial Resolution: This represents the size of the smallest distinguishable unit or area, indicated using standard units, that a variable represents. It determines the level of spatial variation captured by the variable across a dimension. For example, x_spatial_resolution="1000 m", soil_top_layer="0.1 m".
- Grid mapping: This property explicitly defines the coordinate reference system (CRS) used for the spatial coordinate values. This is important because it ensures all geospatial data across different models/modules can be interpreted correctly when compared.
- Vertical datum: Definition of the vertical datum used to describe height above reference ellipsoid.

Metadata is also needed to explicitly define the role or type of the framework exposed BMI variable. The different roles may include:

- *Physical constants*: This refers to known constants, such as the latent heat of vaporization, the gravitational acceleration, etc.
- *Forcing data*: This refers to data that drive the model. These are inputs to the model. For example, precipitation is forcing data for an uncoupled hydrology model.
- *Geophysical attributes*: These variables represent measurable static physical characteristics, such as elevation, slope, albedo, soil and vegetation properties, etc., of the model domain.
- *Landscape type*: This is a categorical property, represented as a named integer, that indicates the landscape type. Landscape types include soil, vegetation, lake, snow, etc. (e.g., the soil hydraulic conductivity is related to the soil landscape and the interception capacity of the vegetation is related to the vegetation landscape). Using landscape types requires enumerating possible landscape types.
- *Landscape class*: This is a categorical property, represented as a named integer, that indicates the landscape class for a given landscape type, e.g., soil type (e.g., silty loam) and vegetation type (e.g., deciduous broadleaf forest). Using landscape classes requires enumerating possible landscape classes.
- *Parameter*: This refers to a variable that influences the output of a model but is not directly predicted or simulated by the model. Parameters can be classified into two main groups with respect to the temporal variation: time-invariant and time-varying. The former indicates that the parameter remains constant during the simulation, whereas the latter indicates the parameter changes over time. Parameters can also be classified as spatially constant and spatially variable. Parameters are often estimated from geophysical attributes (e.g., the porosity and hydraulic conductivity of the soil is estimated from soil properties).
- *Prognostic (state) variable*: This refers to a variable that the model explicitly predicts or simulates over the simulation period. For example, soil moisture. Prognostic state variables are commonly exchanged between programs.
- *Diagnostic variable*: This refers to variables that are derived from state variables or forcing data, and not directly predicted by the model. For example, the saturated area of a model element.
- *Flux*: This refers to variables that represent the rate of mass, energy, or momentum transfer across a boundary. For example, surface runoff. A flux is a special case of a diagnostic variable. Flux variables are commonly exchanged between different models.
- *Algorithmic control parameters*. This refers to parameters that control the behavior, accuracy, and smoothness of the numerical solution (e.g., maximum number of iterations, temporal truncation tolerances, thresholds to merge or subdivide snow layers, etc.).
- *Index variables*: Provides the framework with indices (e.g., index of soil layer, loop counts in iterative steps).
- *Model options*: This refers to the choice of a specific process parameterization (e.g., defined by boolean flags or named integers).
- *Logical variables*: These variables have only “true” or “false” values, and communicate run-time conditions between the model and the framework. Useful for communicating internal stability or numerical error conditions that allow model developers to communicate abnormal module/model behavior to the NextGen Framework. Also

useful to indicate unique state conditions, such as the presence of surface water or snow.

4.2 Time information

We use the following definitions for model/module time steps:

- The framework time step, or data window, is the time interval of the forcing data at the system boundaries. This time step is typically fixed.
- The module time step (minimum, maximum,design_optimal) is the time step for a given module. The module time step may be shorter than the time interval of the forcing data to ensure computational stability and numerical accuracy. The base timestep of each module, as far as the NextGen framework is concerned, is handled by the BMI `update()` function, which moves the module forward by that single specified time step.
- The coupling interval (min/max) is the time interval for a given model component (process module), e.g., as defined using update_until() function. Many BMI implementations include an option for "Update model by a fraction of a time step"[1], which aids in the coupling of models designed to run on different time steps, but many models might not be suitable for every model. For example, CFE includes an hourly GIUH, which relies on the update of a full time step to track the water mass in the "runoff queue", and a fraction of a full 1-hour time step could lead to mass balance errors.
- The module internal time step (min/max) is the time step used internally in a module to ensure computational stability and numerical accuracy. The module time step may be shorter than the module internal time step.

Module time steps should be specified in the BMI metadata, and available through the `get_time_step()` function[2]. If the module internal time step is lower than the specified time step, the module may use any sort of time stepping mechanism so long as the specified time step is hit during the `update()` function.

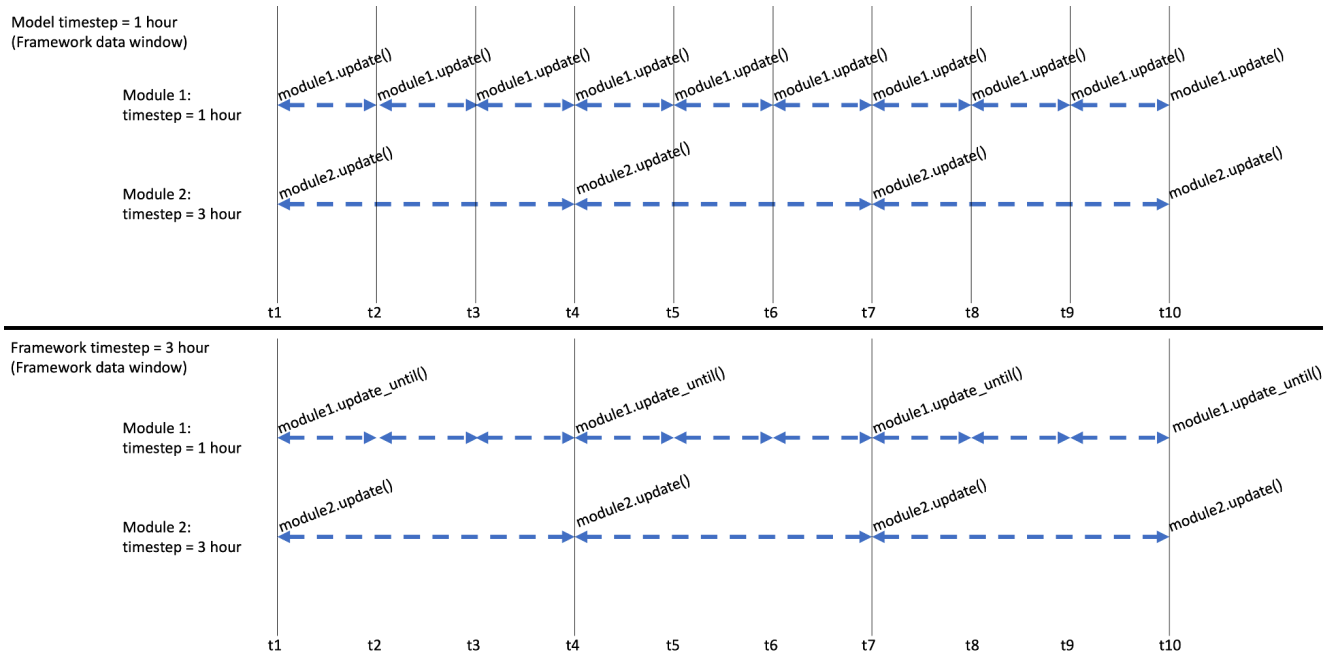
The dynamic time information required includes

- Information on time_bounds, to specify the start and end of the model time step (data window). It is assumed that model output is only provided at the level of framework time steps (data windows).
- The coupling frequency, i.e., the number of times information from a module is passed to other modules per time step. The coupling frequency is dynamic in controlled operator splitting implementations (monitoring and control of splitting error).
- The number of internal module time steps per framework time step (data window). The internal module step size is dynamic if the module includes monitoring and control of temporal truncation error.

---

[1] CSDMS BMI examples: https://github.com/csdms/bmi-example-python/blob/b0a471c2f364a66f167b3209203a6991f39d97f2/heat/bmi_heat.py#L57

[2] CSDMS BMI examples: https://github.com/csdms/bmi-example-python/blob/b0a471c2f364a66f167b3209203a6991f39d97f2/heat/bmi_heat.py#L318



This figure shows an example of running the framework with two different modules at two different framework time steps. In the top example, the framework is run at the native timestep of module 1, where the update function is called every hour for module 1 and every three hours for module 2. The bottom example shows the framework timestep at the native timestep of module 2, the coarser timestep. In this circumstance, module 1 must be run with the `update_until()` function, during which time the module updates internally at its own native timestep of 1 hour, only broadcasting to the framework at the 3 hour interval.

4.3 Calibration parameters

The information on calibration parameters includes:

- name (model, standard, and long) plus definition and units of all parameters
- min/max for all parameters
- default values for all parameters
- list of the subset of parameters that are typically adjusted in calibration

It is also useful to define parameters in a calibration strategy

- any spatial regularization strategy (e.g., parameter multipliers, calibrating coefficients in transfer functions, etc.)
- cost function for optimization (SSE, KGE, transformed flows, etc.) -- to fit multiple objectives (e.g., high flow, low flow)
- calibration algorithm

5. NextGen applications: Transparency and reproducibility of model workflows

Ensuring transferability and reproducibility requires specifying details of the model workflows, forcing data, and boundary conditions.

5.1 Model setup workflow

Model setup workflows are provided by model developers. They should include:

- The minimum specification required for execution (tools, libraries, versions, etc.).
- All code dependencies for packages installed from pip, or conda should be listed in a requirements.txt file. Specific versions required for code compilation should be listed. Dependencies (linux distro requirements, outside compiled software) should be defined in the README.md or in the docs.
- Community models, with active development and usage, are required to use programming languages and libraries that are supported by their respective open-source communities.

The workflow script should include

- name, location/path
- workflow elements / workflow modularity
- datasets used for reproducing the results in published experiments
- re-gridding and upscaling operations
- subset/geographic coordinate limits/constraints
- transformations (e.g., pedotransfer functions)
- cleanly separate model-specific and model agnostic aspects of the workflow

The model setup workflow also requires a list of arguments for the model setup workflow script, e.g., name(s) of required input geospatial or other data from the data library (note: model setup workflow code(s) includes the rules/logic required to estimate parameters and initial states). This can include:

- vertical sub-domain (veg, soil, aquifer, ...)
- datasets that are used to characterize the properties of the sub-domain
- transfer functions that used to transform geophysical attributes to model parameters (e.g., LAI-> interception capacity; sand/silt/clay -> hydraulic conductivity)
- re-gridding of geophysical attributes/parameters (e.g., native spatial polygons [grid, soil unit] to hydrofabric basin of sub-basin "tile")
- upscaling operator (mode, harmonic mean, arithmetic mean, etc.)

It is important to define the order of operations in the model workflow and how the model setup workflow can be coupled to parameter estimation strategies. For example, in a calibration loop, the model setup workflow would run in a replacement mode to enable the parameters in the model setup workflow (e.g. pedotransfer function parameters) to be estimated as part of model calibration.

5.2 External forcing data, initial and boundary conditions

The required external fluxes and states, initial and boundary conditions, defining the state of the system at the limits of the area being modeled, should be documented using standard variable names such as https://csdms.colorado.edu/wiki/CSN_Searchable_List. This is required to provide transparency about how the model/module operates and the assumptions behind it.

- External forcing/driving data: Refers to model/module inputs that drive the model. These can be represented as a list of model variables expressed with the metadata described in section 2.2. Additionally, the forcing source that was used for model calibration needs to be defined. For example:

  forcing_value: “High-Resolution Rapid Refresh”

forcing_value_calibration: “NOAA Analysis of Record for Calibration”

- Initial Conditions (IC): Specifies the prescribed conditions for the model state variables, derived from existing reanalysis or real-time products, that are used at time 0 of the simulation. The metadata for initial conditions needs to be defined separately for each state variable. For example:

  initial_value_source: “GCOS Reanalysis Dataset”

  It is also necessary to specify the datasets that are assimilated into the framework to update model state variables. Since these datasets may not map directly to model states (e.g., fractional snow covered area), the list of datasets used for data assimilation need to be defined as a global attribute.
- Boundary conditions (BC): Specifies the type of boundary condition applied, along with a description of the selected condition and its underlying assumptions.

  boundary_condition_type: “Free Drainage Boundary”

  boundary_condition_description: “Water reaching the bottom of the modeled soil column (often the lowest of four soil layers) is assumed to drain freely out of the model domain”

  boundary_condition_implication: “This assumes that any excess water is removed from the system, simulating an infinite aquifer below the model domain.”

5.3 Model restarts

Many applications require restarts, where all variables that are needed to state-save and hot-start a model are exposed to the framework through the BMI interface. Incorporating restart capabilities enables pausing and resuming of applications seamlessly, ensuring continuity despite daily HPC computing limits, system interruptions, or unexpected failures. Restart capabilities also enable forecasting applications. Minimally, each of these variables should be known to `get_value`, `get_value_at_indicies`, `set_value`, `get_var_type`, `get_var_itemsize` and `get_var_nbytes` in the BMI implementation.

Although important, restart capabilities are not necessary to run a model or couple models in NextGen. As such, restart capabilities are optional and should follow more important requirements such as exposing necessary input and output variables to accept forcing data via the framework and pass output data to other models.

6. Discussion

Several critical challenges remain unaddressed to increase the applicability of the NextGen framework. Key challenges include improving the model representation of observed processes, improving modularity and interoperability, and improving the numerical implementation.

6.1 Improving the model representations of observed processes

Nextgen offers the flexibility to “cherrypick” process algorithms from different hydrologic models – for example, combining a snow module from model A with a vadose zone module from model B – enabling users to build models from modules. These capabilities enable NextGen users to configure models as a heterogeneous mosaic of alternative algorithmic

formulations that faithfully represent the dominant hydrologic processes across different parts of the modeling domain (e.g., a different set of algorithms may be used to simulate the diverse hydrological processes in the glacier-fed basins in Alaska, the pothole landscapes in North Dakota, the tile-drained Midwest, and the karst landscape in Kentucky). Such flexibility in the NextGen framework — i.e., region-specific modeling configurations — requires care to ensure that the region-specific models in NextGen align with our perceptual understanding of hydrological processes, especially given gaps in our process knowledge across different corners of the continent.

To improve the realism of NextGen instantiations, there is a need for a model checker that inspects the proposed formulation for completeness, consistency, and use of appropriate modules.

- As a first set of capabilities, the user should be able to identify processes from a predetermined list and check that these processes are represented in the constructed model.
- Ultimately, it will be important to provide a guide to the user on the dominant processes that are expected in a given location.

The model checker desired here will require a model definition file (e.g., see the initial example using CFE in Appendix E) that contains all information required by the NextGen Framework to determine what a module does, run in terms of order, setup workflow, understand its design and constraints. The information in the model definition file will provide the information required by a perceptual model checker, ensuring that models are being appropriately coupled in space, time, and process (both in terms of design and capabilities).

Defining the processes simulated requires a hierarchical system of standard names (e.g., McMillan, 2022). The list of processes simulated should be linked to the vertical subdomains within the system, with sub-domains defined as a homogenous mixture of constituents; for example:

- vegetation: stems, leaves, liquid water, ice, air
- snow: liquid water, ice, air
- soil: soil particles, liquid water, ice, air
- organic soil: organic matter (e.g., moss, leaf litter), liquid water, ice, air
- lake: liquid water, ice

Each subdomain may be further discretized into control volumes (e.g., soil layers) in order to represent the vertical gradients in model state variables (e.g., matric head, temperature) that drive model fluxes. Each subdomain may have multiple state variables. A general taxonomy for processes may be derived by defining the processes at the boundaries of each subdomain -- at the top, the bottom, and the side -- e.g., for soil, representing infiltration, drainage (recharge), and interflow. The description of spatial variability of each basin in the hydrofabric is a decision made by the modeler.

Since the definition of processes is hierarchical, defining the set of dominant processes requires specifying the model complexity for a specific application; for example:

- To simulate evapotranspiration, the user may decide to explicitly simulate canopy evaporation, transpiration, and soil evaporation, but then the user may opt to use empirical methods to simulate the components of evapotranspiration (e.g., potential ET scaled by water storage on the vegetation canopy and soil moisture).

- Some models may include specific components for e.g. canopy interception, forest litter retention, surface depression storage, etc., whereas others may simply have a "surface storage" component that implicitly includes any of these.
- There will also be differences between explicit and implicit processes. For example, a model may explicitly track canopy storage and evaporation as time series, or simply subtract some fixed value from incoming precipitation to implicitly model canopy interception & evaporation. Such implicit processes are not tracked as time series and would complicate external balance checks.

These efforts to increase the realism of NextGen instantiations will strengthen the link among algorithms, theory, and observations, improve our understanding of the impact of model simplifications, increase the fidelity of model simulations, and hence increase our confidence in model predictions.

6.2 Improving modularity and interoperability

Interoperability requires defining standards and scope for NextGen components so that multiple components can work well together in an integrated interoperable modeling framework. Specifically, mixing-and-matching process algorithms from multiple existing models requires defining standards for process granularity so that it is possible to build models from modules. Just as Lego pieces have standards in terms of their width, length, and height dimensions, model components in NextGen should have standards in terms of their scope (i.e., the coverage of processes). An expanded BMI controller that can incorporate alternative process representations and discretizations can make the model more scalable and applicable in a range of settings while still serving as a testbed for process evaluation and operational use.

The best practice for developing interoperable modules is to ensure consistency in the scope of process modules. One approach is to follow a hierarchical design -- using building blocks of flux calculators and state integrators for different state equations (e.g., mass, energy) and different vertical sub-domains (e.g., vegetation, snow, soil, aquifer). Taken together, these modules are intended to represent the dominant processes in a vertical column (or columns) for a “tile” (or tiles) within a basin in the hydrofabric. The practical constraint in enforcing a common modular construction is the effort required to re-factor the domain science code. If resources are not available to standardize model building blocks, then the model coupling problem entails handling cases where *the same physical process is calculated from more than one model component*. Addressing this issue requires defining which model component is "responsible" for computing specific processes (can be done with BMI get/set functions) and potentially modifying the domain science code for individual process modules to include directives that skip the process computations that are not needed. Since the lack of standardized modules can result in an unwieldy coupling solution, dedicated effort to better define model building blocks will improve the applicability of multi-modeling frameworks.

6.3 Improving the numerical implementation

Implementing models in the NextGen framework requires attention to numerical errors at multiple levels of process granularity. A common use case in NextGen is to run a set of modules – mod1, mod2, modN – in a prescribed sequence, with state updates determined by each individual module. This leads to two potential issues:

- Since we cannot assume that a given process module is numerically accurate, we need to evaluate and potentially control the temporal truncation error.
- Since we cannot assume that an arbitrarily defined sequence of process calculations will provide accurate answers, it is important to monitor and control the splitting error.

Improving the numerical implementation has a tradeoff between accuracy and efficiency -- i.e., more accurate numerical methods typically require longer computational runtimes.

6.3.1 Numerical errors in individual process modules

The numerical method used in the state integrator for a given process subset may have substantial numerical errors. For example, some process modules that are included in the NextGen framework may have an ad-hoc numerical implementation, where the state updates are intertwined with the flux calculations (e.g., an uncontrolled form of operator splitting that sequentially add fluxes to a model state variable, updating the state variable after each flux is added, and using the updated state variable for each new flux calculation). Other process modules may have substantial numerical errors because they use explicit (forward) Euler methods to update model states on daily time steps. Other process modules may be more numerically robust, e.g., by using adaptive sub-stepping methods to satisfy user-specified tolerances for temporal truncation errors, or by using implicit (backward) Euler methods that can provide more robust solutions on longer time steps. The numerical error depends on the initial model state, the model forcing data, and the model parameter set.

It is important to check that the temporal truncation error is satisfactory for each model component. This can be done by developing functionality to check that a model component provides similar solutions on shorter time steps, e.g., successively halving the time step until the difference in solutions meet user-prescribed tolerances. This can be done either (a) as a pre-processing step to define a suitable time step for each individual process module; or (b) as an implementation at run time (in each time step) to dynamically adjust the step length in individual process modules.

6.3.2 Numerical errors in the coupled solution

The use case in NextGen to run a set of modules in a prescribed sequence is an uncontrolled form of Operator Splitting. In comparison to the approach where all state variables are solved simultaneously, this uncontrolled operator splitting approach can lead to splitting errors (i.e., the results depend on the order of operations) when processes are tightly coupled.

It is possible to monitor and control splitting errors using higher order splitting methods (e.g., Strang splitting) that use fractional time steps for the different process modules. Consider the case with two process modules – initialize mod 1 with the state at the end of the previous time step, run mod1 for half the time step, initialize mod2 with the end state from the half-step run from mod1, run mod2 for the entire time step, initialize mod1 with the end state from mod2, and run mod1 for half the step. The difference between the two mod1 simulations can be used to define the splitting error – if the splitting error is below a user-prescribed tolerance, then the control program should reduce the length of the sub-step. A general implementation of monitoring/controlling splitting errors is straightforward—for example, it is possible to implement a nested loop, with the outer loop the sub-steps, i.e., continue until reach the end of the time step, cycling to account for splitting error, and an inner loop, i.e., fractional steps and operators where the *i*-th iteration of the loop runs operator *j* for fractional step alpha[*k*].

The practical motivation for implementing higher-order operator splitting methods is to improve the numerical solution without major refactoring of component modules or major changes to the NextGen framework.

Another approach to reduce splitting errors is to tightly couple model components (e.g., a hub-and spokes model that combines modules as part of a general solver). For example, Hughes et al. (2022) uses a hierarchical approach that refactors the MODFLOW source code to map onto the main BMI control functions (initialize, update, finalize) at different levels of granularity. The traditional BMI functionality is the main model control loop for MODFLOW that initializes the model, updates the model for a given time step, and finalizes the model simulation, with the time step loop contained in the driver program that calls the BMI control functions. The update step is then also mapped onto the BMI initialize, update, finalize functions to prepare the time step (initialize), calculate the time step (update), and finalize the time step (finalize). This modularity provides opportunities to interactively modify the time-varying data. At the finest level of granularity, the control function to calculate the time step is also mapped onto the BMI control functions to prepare the solver (initialize), solve the nonlinear system of equations (update) and compute additional variables based on the converged solution (finalize), with the iteration loop in the driver that calls the BMI control functions (i.e., the update is for a single iteration). The advantage of using BMI control functions for the solver is that it provides BMI control within the iteration loop -- this makes it possible to tightly couple multiple model components and have the coupled solution converge at the outer iteration level, i.e., within a single time step. As noted by Hughes et al. (2022), this fine-grained approach is preferable to the loosely-coupled sequential alternative because it avoids temporal oscillations in the solutions and results in a simulation that is less sensitive to the prescribed sequence of operations.

6.3.3 Solver metadata

The information on the solver should include:

- The numerical method used in the state integrator for a given process subset (e.g., soil hydrology, aquifer, etc.), e.g., explicit (forward) Euler, implicit Euler, any adaptive sub-stepping methods to satisfy user-specified tolerances for temporal truncation errors.
- Operator splitting approximations (e.g., the prescribed order of operations), and if the coupling frequency is adjusted to satisfy user-specified tolerances for splitting errors

The information on the solver should be defined per computational unit (e.g. catchment divide in the hydrofabric) with options configurable in the model realization file and Multi-BMI formulation definition therein.

7. Summary

The motivation for developing the NextGen framework is that a well-designed and frequently updated community model can capitalize on the economies of scale and increase the benefits of hydrologic-water resource modeling for society. The value proposition of NextGen is that it is possible to increase the efficiency and effectiveness of hydrologic modeling by improving modularity (e.g., separating concerns so that model developers can more easily work on the area of the code where they have the most expertise, reducing costs of model development and maintenance). The value proposition is also to increase interoperability (e.g., a clean

modular model design can make it easier to share code across different model development groups). Moreover, the value proposition of NextGen increases the applicability of community models by advancing the tools in the model "ecosystem" (e.g., tools to configure the community models in new settings, tools for parameter estimation, tools for model evaluation; and tools for visualization) as well as increasing the applicability of NextGen through improvements in scientific validity (e.g., narrowing the gap between models and theory) and improvements in model accuracy (e.g., by having the flexibility to tailor the model structure to suit specific applications in specific regions). The development of the NextGen prototype provides the initial computational capabilities that enables different research and operational groups to collaborate more effectively, contributing their unique data, information, and knowledge to advance community modeling capabilities.

This document defines technical standards to support the development of the NextGen framework, as well as standards and guidelines for the development and application of models intended to be coupled within the framework. Defining these technical standards helps ensure that the minimum set of variables are exposed through the BMI interface to the NextGen Framework to enable full functionality of coupled NextGen instantiations. These technical specifications promote consistency in the application of the BMI standard in a way that provides maximum flexibility and information required for its successful use within the NextGen Framework.

Epilogue

Developing NextGen specifications is an ongoing process led by CIROH Working Group 2 on community water modeling. We welcome contributions to advance NextGen standards. If you are interested in contributing, then please contact NOAA/NWS/OWP Chief Scientist Fred Ogden (fred.ogden@noaa.gov) or WG2 Co-Chairs Mukesh Kumar (mkumar4@eng.ua.edu) and Martyn Clark (martyn.clark@ucalgary.ca).

Appendix A. BMI functions

BMI provides a set of functions that can be broken unto the following categories:

- Model information functions
- Variable information functions
- Time functions
- Model control functions
- Variable getter and setter functions
- Model grid functions

A.1 Model information functions

The basic functions in this category provide fundamental model information, such as input and output variable names. The names provided here facilitate model coupling and output writing. The best practice is to We recommend mapping model variable names to CSDMS standard names for easy interpretation.

A.2 Variable information functions

These functions return information about the variables a model exposes through BMI. This includes the variable's units, data type, and memory requirements. For NextGen's automatic unit conversion to work, variable units must be compliant with the UDUNITS syntax.

A.3 Time functions

Time functions provide the current, start, and times of a model simulation along with the model's time step. NextGen uses these functions to keep the simulation clock across various model's in an instance's stack.

A.4 Model control functions

Your model, as noted above, must follow the initialize, update, and finalize paradigm to work in NextGen. The corresponding BMI functions allow NextGen to initialize multiple model instances and set parameter values, execute a given model for one time step and advance the model clock forward in time, and close any open file connections and free memory.

A.5 Variable getter and setter functions

These functions enable NextGen to change model variable values and pass data from one model to another. All forcing variables in a given model must be exposed via the set_value function so that the forcing engine can update these values each time step. Similarly, all variables that will be passed to an additional model in a model stack or written as output must be exposed via get_value.

A.6 Model grid functions

The grid functions describe a model's spatial discretization along with its parameters (e.g., the origin and spacing of a uniform rectilinear grid).

A.7 Minimal BMI implementation

The BMI provides many functions which may not be applicable to a given model or module. There is, however, a minimum set of BMI functions which the NextGen model engine expects to be implemented. The following is a list of these functions which require proper implementations:

initialize
update
finalize
get_component_name

get_intput_item_count
get_input_var_names
get_output_item_count
get_output_var_names

set_value
get_value
get_values_at_indicies
get_value_ptr*

get_var_grid**
get_var_itemsize
get_var_nbytes
get_var_type
get_var_units

get_start_time
get_end_time
get_current_time
get_time_units
get_time_step

This list represents the functions NextGen expects to interact with meaningful implementations at the time of this writing. It may change over time.

* Not currently applicable for Fortran modules
** May just return 0 for scalar/non-grid implementations

Appendix B. Registration functions

The NextGen model engine requires one or more additional non-BMI function implementations for C, C++, and Fortran modules. These functions are necessary for the model engine to dynamically load the module into the simulation runtime.

B.1 C Registration

Modules written in C need to provide in their library interface a registration function. While other names can be configured for this function, the model engine will by default look for this function in the library:

```
C/C++

Bmi* register_bmi(Bmi *model);
```

In most typical C development, this function signature will be in a header file, and implemented in a source file. The registration function takes as an input a Bmi struct pointer, and it must set the function pointers of that struct to the BMI functions defined for the module:

```
C/C++

Bmi* register_bmi(Bmi *model) {
    if (model) {
        ...
       model->initialize = Initialize;
       model->finalize = Finalize;
       model->Update = Update;
        ...
    }
}
```

B.2 C++ Registration

Modules written in C++ need to provide in their library interface a model create and destroy function. While other names can be configured for these functions, the model engine will by default look for these functions in the library:

```
C/C++
extern "C"
    {
      /**
```

```
        * @brief Construct this BMI instance as a normal C++ object, to be
returned to the framework.
      * @return A pointer to the newly allocated instance.
      */
      MyBmiModelClass *bmi_model_create()
      {
        /* You can do anything necessary to set up a model instance here, but
do NOT call `Initialize()`. */
             return new MyBmiModelClass(/* e.g. any applicable constructor
parameters */);
      }

      /**
        * @brief Destroy/free an instance created with @see bmi_model_create
        * @param ptr
        */
      void bmi_model_destroy(MyBmiModelClass *ptr)
      {
        /* You can do anything necessary to dispose of a model instance here,
but note that `Finalize()`
         * will already have been called!
         */
        delete ptr;
      }
    }
```

Note that these functions must be in an extern “C” code block.

B.3 Fortran Registration

Modules written in Fortran need to provide in their library interface a registration function. While other names can be configured for this function, the model engine will by default look for this function in the library:

Unset

```
function register_bmi(this) result(bmi_status) bind(C, name="register_bmi")
      use, intrinsic:: iso_c_binding, only: c_ptr, c_loc, c_int
      use MYMODULE
      implicit none
       type(c_ptr) :: this ! If not value, then from the C perspective `this`
is a void**
      integer(kind=c_int) :: bmi_status
      !Create the model instance to use
```

```
        type(MY_BMI_TYPE), target, save :: bmi_model !should be safe, since
this will only be used once within scope of dynamically loaded library
      !Create a simple pointer wrapper
      type(box), pointer :: bmi_box

      !allocate the pointer box
      allocate(bmi_box)
      !associate the wrapper pointer the created model instance
      bmi_box%ptr => bmi_model
      !Return the pointer to box
      this = c_loc(bmi_box)
      bmi_status = BMI_SUCCESS
end function register_bmi
```

Note: you should only change the MYMODULE and MY_BMI_TYPE in this function.

B.4 Python Registration

There are no additional registration requirements when using Python BMI modules.

Appendix C. Specifications for mass conservation

The following yaml file format will allow the Framework to perform the calculation of Eqn. 1 for conservation of mass or energy over arbitrary control volumes, with an arbitrary number of control surfaces, for an arbitrary number of flux variables.. Conservation of momentum will require the addition of vector components for each control surface (outward normal unit vector and flux direction vector) to the yaml definition.

max_num_control_volumes: *n*

control_volume_plan_area_m2: [11.32, 78.13, … A_n]

control_volume_thickness_m: [2.0,2.0,2.0,...T_n]

control_volume_m3: [] // leave brackets empty if area and thickness specified.

Alternatively, populate and leave area and thickness empty

storage_var: *storage_variable_name*

prev_storage_var: *prev_storage_var_name* // module must store & expose the previous value of storage.

max_num_control_surfaces: *k*

max_num_flux_vars: *i* // NOTE: i>1 if conservation depends on more than one flux variable name. For this example i=3, `lateral_flow`, `infiltration`, and `percolation`.

flux_var_name: flux_var_name_1 // example `lateral_flow`

control_volume_number: 1

[1,-6] // change in storage in c.v. 1 depends on two adjacent lateral flow fluxes with indexes of 1 and -6, which means flux(1) and -flux(6)

control_volume_number: 2

[-1,2]

control_volume_number: 5

[5,0] // the storage in this control volume depends on only one lateral flow , flux 5.

control_volume_number: *m*

[-5,6]

flux_var_name: flux_var_name_2 // example `infiltration`

control_volume_number: 1

[1]

…

control_volume_number: m

[m]

flux_var_name: flux_var_name_3 // example `percolation`

control_volume_number: 1

[1]

…

control_volume_number: m

[m]

This information will allow the framework to solve the conservation equation across an arbitrary number of control volumes, each with an arbitrary number of flux variables. Note that this section can be nested to solve different conservation laws (mass, energy, momentum).

Appendix D. Coding standards

Code should be easy to understand, avoid complicated or obfuscated solutions whenever possible. The following guidelines should be followed where possible.

Variable names:

- Use self-describing variable names, even if they have a large number of characters. For example, canopyEvap is preferable to ce
- Define every single variable, including units, with each variable on a separate line. For example, in fortran, use
  real(dp),intent(out) :: canopyEvap ! canopy evaporation (kg m-2 s-1)

Hard-coded numbers, physical constants and parameters:

- Do not use hard-coded numbers in equations to represent physical constants or model parameters.
- All physical constants are defined in a single shared module and should *not* be redefined at multiple points in the code. If you add physical constants, they must be added to the shared module.
- All model parameters are ideally read in from model configuration files to allow model users to experiment with parameter values without requiring code changes. During model development, it is sometimes easier to skip this step. In that case, at the very least, model parameters must be identified at the top of a routine as part of the variable definition section, e.g. in fortran, use
  real(dp),parameter :: facTrustDec=0.25_dp ! factor decrease in trust region

Commenting:

- Include a comment block at the start of each subroutine or function, providing a brief description of what the subroutine does (including references).
- Do not commit code with large sections of code commented out. A version control system should be used for tracking different versions. There is no need to do it manually by commenting out code, which is much more likely to lead to confusion. The only exception is for some debug statements that may be uncommented and reused during debugging.
- Comment often. Strive for a meaningful comment to complement every logical block of code. Describe the purpose of the code, e.g. "why" not "what". It takes much less time to "*comment as you go*" than try and comment afterwards.
- When there is potential for confusion, define units for the left-hand-side of the assignment
- Use a consistent commenting style. (this could be something that is decided for the entire NextGen system)
- Don’t leave TODO comments in the code.

Appendix E: Model Definition File

The Next Generation Water Resources Modeling Framework (NextGen) is a set of standards based, model agnostic, model interoperability software tools. The design of the NextGen framework allows users to couple process models as modules to construct a larger model formulation that the user believes will be more performant, and provides options for calibration and evaluation of different model formulations.

The NextGen Framework applies the Basic Model Interface (BMI) standard for model coupling. The BMI standard is a set of 40+ functions that allow initialization, querying, substitution, and finalization of a model/module. Each model/module uses its own internal discretization and solvers. The BMI standard allows querying a model to learn about its discretization. In the BMI standard, models/modules running using the BMI standard are unaware of other models/modules running. This requires an overarching control program to take care of “orchestrating” execution of these independent models and modules.

There are very few complete hydrologic models that solve for all relevant fluxes and model states, particularly at larger scales. Most models actually focus on a particular subset of the hydrologic cycle, which is a good thing, because the permutations of dominant processes in hydrology is vast. Dominant processes vary tremendously by factors such as: land cover and land use, climate, soils, surficial geology, season, vegetation, antecedent rainfall, rainfall rate, reservoirs, diversions, irrigation, etc. In a particular catchment the dominant process can change depending on changes in these factors, which gives rise to threshold behavior.

The NextGen framework treats every model regardless of its complexity as a module. Some codes are simple modules, such as adapters that read forcing input data or perform a single function such as streamflow data assimilation. In the NextGen framework, modules are linked together to create “formulations”. The instructions that tell the NextGen model engine how to create a formulation, and the order to execute the different models is contained in what is called the “model realization file”.

The BMI standard includes querying ability to learn about a module. These functions allow the orchestration program to know the discretization and exposed state variables and their units, these functions do not reveal all the metadata required to check for formulation completeness and appropriateness.

However, the NextGen framework design includes a number of unique features:

1. Model Checker: Compares user constructed formulations against a user developed perceptual model to check for model completeness. The model checker seeks to answer the question “does the constructed formulation solve for all the hydrologic state variables and fluxes that the user thinks are important?” The model checker also verifies that selected modules are compatible with each other in terms of their design, and can run adapters to estimate missing fluxes or state variables.
2. Execute models written in different programming languages. So far, C, C++, Fortran, and Python are included.
3. Run model formulations on individual workstations/laptops and distributed memory computers using MPI.

4. Couple model forcings to hydrologic modules that use different discretizations.
5. Run different models in different parts of the domain, together.

Because the BMI standard lacks module metadata required for constructing formulations from modules, the NextGen Framework requires provision of this information that is provided by domain science experts. This information is stored in the “model definition file”. The model definition file contains all information that the framework needs to setup, check appropriateness and completeness, and link a model/module to other models/modules in the framework. The model definition file contains no information for a particular hydrologic simulation because that information belongs in the model realization file. The standard definition for units are used from the unix/linux “units” package, as defined in the file /usr/share/units/definitions.units

```
ModelMetaData:
      name: CFE
      type: Catchment rainfall-runoff model
      version: 1.0
      description: "The conceptual functional equivalent (CFE) to version 3.1 and
earlier of the WRF-Hydro based National Water Model, which is catchment based rather
than gridded."
      programming language: C # The programming language used to implement the model
      source: https://github.com/NOAA-OWP/cfe # Link to the source code repository

# TemporalDesign specifies the time-related aspects of the model, including
# the time unit (e.g., seconds, minutes, hours), and the optimal, minimum,
# and maximum time steps the model can handle.
TemporalDesign:
      timeUnit: hour  # Defines the time unit for the model's simulation steps -
                       #<second,minute,hour,day,pentad,week,month,other>
      timeUnitOther: <blank if not used> #Used if the time unit is not a standard
unit
      timestepOptimal:  1 #The optimal time step in hours for running the model
      timestepMin: 1     #The minimum time step the model can handle
      timestepMax: 1     #The maximum time step the model can handle

# SpatialDesign defines the spatial aspects of the model, including units of length
# and area, and the minimum and maximum spatial extents the model can simulate.
SpatialDesign:
      lengthUnit: m # Defines the unit for spatial dimensions -
<mm,cm,dm,m,km,in,ft,yd,statutemile,other>
      lengthUnitOther: <blank if not used> # Used if the length unit is not a
standard unit
      areaUnit: m2 # Defines the unit for area calculations -
                    # <mm2,cm2,dm2,m2,ha,km2,in2,ft2,acre,statutemile2,other>
      areaUnitOther: <blank if not used> # Used if the area unit is not a standard
unit
      areaMin: 1 # <integer> The minimum area the model can simulate (in units above)
      areaMax: 1 # <integer> The maximum area the model can simulate (example value)

# ProcessSimulated defines the specific hydrological processes the model simulates.
# Each process includes a name, description, dimensionality, and the optimal,
# minimum, and maximum timesteps and spatial steps the model can handle.

################## Example format for calculated states or fluxes #############
Calculates:
       name: <see list produced in paper by Hillary McMillan et al.>
```

```
       type: <state>, <flux>
      description: "Text describing this state or flux"
      dimensionality: (0,1, 2, 3) !note 0=point, 1=1D, 2=2D, 3=3D
        timestepOptimal:  num
        timestepMin: num
        timestepMax: num
      spaceStepMinDim1: num  ! blank if dimensionality==0 this is the minimum
discretization of spatial dimension 1.
      spaceStepOptDim1: num  ! ... this is the optimal or target/desired
discretization of spatial dimension 1.
      spaceStepMaxDim1: num ! ... this is the maximum discretization of spatial
dimension 1.
      spaceStepMinDim2: num  !blank if dimensionality<2
      spaceStepOptDim2: num  ! ...
      spaceStepMaxDim2: num ! ...
      spaceStepMinDim3: num  !blank if dimensionality<3
      spaceStepOptDim3: num  ! ...
      spaceStepMaxDim3: num ! ...
            acceptsVar: <stdname>  ! these are variables that this process simulator
will accept updates through the BMI
            acceptsVar: <stdname>  ! ...
            ...
            predictsVar: <stdname>  ! these are variable names that this process
simulator skillfully predicts that other models might want to use.
            predictsVar: <stdname>  !...

################### CFE States and Fluxes calculated starts here #################
Calculates:
    name: Direct Runoff  # Name of the hydrological process simulated
    type: flux
    description: "Surface runoff from rainfall/throughfall input"
    Option: Schaake
    dimensionality: 0  # Dimensionality of the process <e.g., 0 = point, 1 = 1D, 2 =
2D, 3 = 3D>
    timestepOptimal: 1  # Optimal timestep for this process (in time units specified
previously)
    timestepMin: 1  # Minimum timestep for this process
    timestepMax: 1  # Maximum timestep for this process
    spaceStepMinDim1: <blank>  # Minimum spatial step size for 1st dimension
    spaceStepOptDim1: <blank>  # Optimal spatial step size for 1st dimension
    spaceStepMaxDim1: <blank>  # Maximum spatial step size for 1st dimension
    spaceStepMinDim2: <blank>  # Minimum spatial step size for 2nd dimension
    spaceStepOptDim2: <blank>  # Optimal spatial step size for 2nd dimension
    spaceStepMaxDim2: <blank>  # Maximum spatial step size for 2nd dimension
    spaceStepMinDim3: <blank>  # Minimum spatial step size for 3rd dimension
    spaceStepOptDim3: <blank>  # Optimal spatial step size for 3rd dimension
    spaceStepMaxDim3: <blank>  # Maximum spatial step size for 3rd dimension

Calculates:
    name: GIUH Runoff  # Time delayed surface runoff response by GIUH
    type: flux
    description: "Simulates lagged and attenuated runoff using the Geomorphological
Instantaneous Unit Hydrograph"
    dimensionality: 0  # This is a conceptual point-process delay/routing function
    timestepOptimal: 1
    timestepMin: 1
    timestepMax: 1
    spaceStepMinDim1: <blank>  # Minimum spatial step size for 1st dimension
    spaceStepOptDim1: <blank>  # Optimal spatial step size for 1st dimension
```

```
    spaceStepMaxDim1: <blank>  # Maximum spatial step size for 1st dimension
    spaceStepMinDim2: <blank>  # Minimum spatial step size for 2nd dimension
    spaceStepOptDim2: <blank>  # Optimal spatial step size for 2nd dimension
    spaceStepMaxDim2: <blank>  # Maximum spatial step size for 2nd dimension
    spaceStepMinDim3: <blank>  # Minimum spatial step size for 3rd dimension
    spaceStepOptDim3: <blank>  # Optimal spatial step size for 3rd dimension
    spaceStepMaxDim3: <blank>  # Maximum spatial dimension for 3rd dimension

Calculates:
   name: Nash Lateral Subsurface Flow/Runoff
    type: flux
    description: "Simulates lateral subsurface flow through the Nash cascade"
    dimensionality: 0
    timestepOptimal: 1
    timestepMin: 1
    timestepMax: 1
    spaceStepMinDim1: <blank>  # Minimum spatial step size for the 1st dimension
    spaceStepOptDim1: <blank>  # Optimal spatial step size for the 1st dimension
    spaceStepMaxDim1: <blank>  # Maximum spatial step size for the 1st dimension
    spaceStepMinDim2: <blank>  # Minimum spatial step size for the 2nd dimension
    spaceStepOptDim2: <blank>  # Optimal spatial step size for the 2nd dimension
    spaceStepMaxDim2: <blank>  # Maximum spatial step size for the 2nd dimension
    spaceStepMinDim3: <blank>  # Minimum spatial step size for the 3rd dimension
    spaceStepOptDim3: <blank>  # Optimal spatial step size for the 3rd dimension
    spaceStepMaxDim3: <blank>  # Maximum spatial step size for the 3rd dimension

Calculates:
    name: Total Discharge
    type: flux
    description: "Calculates the total discharge from all runoff processes"
    dimensionality: 1
    timestepOptimal: 1
    timestepMin: 1
    timestepMax: 1
    spaceStepMinDim1: <blank>  # Minimum spatial step size for the 1st dimension
    spaceStepOptDim1: <blank>  # Optimal spatial step size for the 1st dimension
    spaceStepMaxDim1: <blank>  # Maximum spatial step size for the 1st dimension
    spaceStepMinDim2: <blank>  # Minimum spatial step size for the 2nd dimension
    spaceStepOptDim2: <blank>  # Optimal spatial step size for the 2nd dimension
    spaceStepMaxDim2: <blank>  # Maximum spatial step size for the 2nd dimension
    spaceStepMinDim3: <blank>  # Minimum spatial step size for the 3rd dimension
    spaceStepOptDim3: <blank>  # Optimal spatial step size for the 3rd dimension
    spaceStepMaxDim3: <blank>  # Maximum spatial step size for the 3rd dimension

Calculates:
    name: Streamflow Volume Flux
    type: flux
    description: "Calculates the volume flux of streamflow"
    dimensionality: 1
    timestepOptimal: 1
    timestepMin: 1
    timestepMax: 1
    spaceStepMinDim1: <blank>  # Minimum spatial step size for the 1st dimension
    spaceStepOptDim1: <blank>  # Optimal spatial step size for the 1st dimension
    spaceStepMaxDim1: <blank>  # Maximum spatial step size for the 1st dimension
    spaceStepMinDim2: <blank>  # Minimum spatial step size for the 2nd dimension
    spaceStepOptDim2: <blank>  # Optimal spatial step size for the 2nd dimension
    spaceStepMaxDim2: <blank>  # Maximum spatial step size for the 2nd dimension
    spaceStepMinDim3: <blank>  # Minimum spatial step size for the 3rd dimension
```

```
    spaceStepOptDim3: <blank>  # Optimal spatial step size for the 3rd dimension
    spaceStepMaxDim3: <blank>  # Maximum spatial step size for the 3rd dimension

Calculates:
    name: Deep Groundwater to Channel Flux
    type: flux
    description: "Simulates the flux from deep groundwater to the channel"
    dimensionality: 1
    timestepOptimal: 1
    timestepMin: 1
    timestepMax: 1
    spaceStepMinDim1: <blank>  # Minimum spatial step size for the 1st dimension
    spaceStepOptDim1: <blank>  # Optimal spatial step size for the 1st dimension
    spaceStepMaxDim1: <blank>  # Maximum spatial step size for the 1st dimension
    spaceStepMinDim2: <blank>  # Minimum spatial step size for the 2nd dimension
    spaceStepOptDim2: <blank>  # Optimal spatial step size for the 2nd dimension
    spaceStepMaxDim2: <blank>  # Maximum spatial step size for the 2nd dimension
    spaceStepMinDim3: <blank>  # Minimum spatial step size for the 3rd dimension
    spaceStepOptDim3: <blank>  # Optimal spatial step size for the 3rd dimension
    spaceStepMaxDim3: <blank>  # Maximum spatial step size for the 3rd dimension

Calculates:
    name: Conceptual Soil Moisture Storage
    type: state
    description: "The storage of water in the 2 m thick soil conceptual reservoir"
    dimensionality: 1
    timestepOptimal: 1
    timestepMin: 1
    timestepMax: 1
    spaceStepMinDim1: <blank>  # Minimum spatial step size for the 1st dimension
    spaceStepOptDim1: <blank>  # Optimal spatial step size for the 1st dimension
    spaceStepMaxDim1: <blank>  # Maximum spatial step size for the 1st dimension
    spaceStepMinDim2: <blank>  # Minimum spatial step size for the 2nd dimension
    spaceStepOptDim2: <blank>  # Optimal spatial step size for the 2nd dimension
    spaceStepMaxDim2: <blank>  # Maximum spatial step size for the 2nd dimension
    spaceStepMinDim3: <blank>  # Minimum spatial step size for the 3rd dimension
    spaceStepOptDim3: <blank>  # Optimal spatial step size for the 3rd dimension
    spaceStepMaxDim3: <blank>  # Maximum spatial step size for the 3rd dimension


<repeat Calculates: as necessary>

# BroadcastedVariables lists the variables that the NextGen Framework can interact
with.
# These include variables the model accepts as inputs and those it predicts as
outputs.
BroadcastedVariables:
  acceptsVar:
    - atmosphere_water__time_integral_of_precipitation_mass_flux  # The model accepts
this variable as an input
    - water_potential_evaporation_flux  # Another input variable
  predictsVar:
    - land_surface_water__runoff_depth  # The model predicts this variable as an
output
    - land_surface_water__runoff_volume_flux  # Another output variable
    - DIRECT_RUNOFF
    - GIUH_RUNOFF
    - NASH_LATERAL_RUNOFF
    - DEEP_GW_TO_CHANNEL_FLUX
```

```
    - SOIL_CONCEPTUAL_STORAGE

# SetupWorkflow defines the scripts and their execution paths required to set up
# and run the model. It also lists the required input data sources that the framework
# must verify before running the model.
SetupWorkflow:
      -   scriptName: <name>
            path:  <path>
            options: <command line options>
            requiredInputs: <a list of the input data sources required to run the
model setup script> ! This is required so the framework can verify the existence of
these data sets in the input library.
  - scriptName: "initialize_cfe"
    path: "./scripts/initialize_cfe.py"  # Path to the script that initializes the
model
    options: "--config ./config/cfe_config.json"  # Command line options for the
script
    requiredInputs:
      - "./data/forcing_data.csv"  # Input data required to run the script
      - "./config/cfe_config.json"  # Configuration file required for initialization

  - scriptName: "run_cfe"
    path: "./scripts/run_cfe.R"  # Path to the script that runs the model
    options: "--run"  # Command line options for running the model
    requiredInputs:
      - "./data/forcing_data.csv"  # Input data required to run the model
      - "./config/cfe_config.json"  # Configuration file required for the run
```

References:

Kolawa, Adam; Huizinga, Dorota (2007). Automated Defect Prevention: Best Practices in Software Management. Wiley-IEEE Computer Society Press. p. 75. ISBN 978-0-470-04212-0.