11 Model Framework Integration

Version 11.12.05, 2016-09-27

TSTool can integrate with modeling frameworks or serve as a modeling framework. This chapter explores various aspects of this integration.

11.1 Modeling Framework Background

Modeling frameworks may run stand-alone using simple file input and output or may use databases for data management. The term "framework" is used to imply that the model is more than simple equations, but encompasses an operational tool that considers data management, model execution, and results visualization, which can be applied in multiple situations (is not a one-off hard-coded model).

Modular modeling components will have different names in each framework, for example "module", "command", "operation", or "model". For discussion purposes, this documentation uses the term "modeling framework" for the entire framework, and "model" for an individual model component. In most cases, each model will include the following features:

- Configuration data:
    - parameters or properties
    - configuration for dynamic data, such as identifiers to find the data
- Dynamic input data:
    - time series
    - spatial data as layers, grids, etc.
    - database tables and other related data
- Model states, which represent the overall state of the model and can be used to restart the model from a point in time, typically represented as:
    - identifier for the model, to locate the states
    - date/time corresponding to when states were saved (and can be read to restart)
    - names of states
    - values for states, corresponding to each name
        - singular values (numbers, strings, boolean values)
        - arrays of values (numbers, strings, boolean values)
- Output:
    - time series
    - spatial data
    - database tables and other data

Each model may run independently of the framework but depends on the framework to provide input/output (I/O) functionality and overall coordination of model execution in the proper sequence (workflow management) in order to provide the integrated functionality of the framework.

TSTool is well-suited for executing a workflow of models (called commands in the TSTool world), specify configuration data as command parameters, process dynamic data as time series, tables, and spatial data. However, TSTool does not provide a build-in framework for managing model states. Instead, TSTool provides features to manage states, where needed, using existing general features such as tables. This allows TSTool to integrate with many modeling frameworks, or serve as a modeling framework.

The remaining sections of this chapter describe how to integrate TSTool with modeling frameworks.

11.2 Options for TSTool and Model Framework Integration

The following are the primary ways that TSTool can be integrated (or implement) a modeling framework. One or more of these approaches can be used to implement an operational modeling framework.

1. Use TSTool to prepare data for a modeling framework, but do not call from the modeling framework.
    a. TSTool is essentially a pre-processor for model input and post-processor for model output.
    b. Communicate using shared database, files, etc.
2. Use TSTool as a model called from the modeling framework.
    a. TSTool adheres to the modeling framework via batch command-line call (`tstool -commands …`).
    b. Communicate using shared database, files, etc.
3. Use TSTool as the modeling framework with no dependency on external tools, implementing more advanced features in commands as necessary.
    a. For example, the `VariableLagK()` command allows model states to be read at the start of execution and saved at the end of execution.
    b. Some high-level control of run dates is implemented, for example, using TSTool `${Property}` syntax to move dates forward over time so that commands can respond accordingly.
4. Use TSTool as the modeling framework with integration to external tools, implementing more advanced features in commands as necessary.
    a. For example, call model framework models using the `RunProgram()` command, with appropriate management of dynamic data and model states using TSTool generic features such as tables and naming conventions.
    b. Some high-level control of run dates is implemented, for example, using TSTool `${Property}` syntax to move dates forward over time so that commands can respond accordingly.

The approach that is implemented in a production environment depends on multiple factors, for example:

- Which framework provides the most core functionality to meet operational requirements?
- Which approach performs the best (execution speed)?
- Which approach provides flexibility to explore various implementations?
- Which approach is the easiest to implement and maintain?

11.3 Model State Management in TSTool

State management involves reading model states before execution in order to initialize states, and then saving the states at the end of the execution so that a future run can be initialized with those values. The general execution sequence is as follows, using the VariableLagK command as an example:

- **Initialize states**. TSTool begins to run the command file. Global standard properties such as `${OutputStart}` and `${OutputEnd}` may be defined to manage the processing period (other date/time objects also may be defined as properties). These properties can be overridden by date/times specified in command parameters. In this example the `OutputStart` parameter indicates when to read states to initialize the model.
- **Write states**. The `VariableLagK()` command uses parameters to indicate when to save states:

- The `StateSaveDateTime` parameter indicates a specific date/time to save states and can be specified using a `${Property}`. For example, save for `${OutputEnd}`. This may be appropriate if running in real-time mode.
- The `StateSaveInterval` parameter indicates that states should be saved at a regular interval. This allows restarting model runs at a point in history. A longer interval may be appropriate for historical runs and a shorter interval for real-time runs.

TSTool does not provide a specific solution for managing model states because it is intended to be a generic tool and commands traditionally have not required state management. However, as TSTool functionality has increased, opportunities for integration with modeling frameworks has also increased. For example, the increased support of processor `${Property}` allows flexibility in configuring workflows and integration with tables from databases, spreadsheets, and files provides a convenient way to manage model state data.

A general approach for handling model (command) states is to use a table to represent the states. The state table can then be read from and written to any number of persistent formats that are supported by TSTool, such as comma-separated-value (CSV) and Excel. Commands that require state management can then utilize the state table as input and output. In this approach, TSTool provides functionality for state management but does not define strict requirements for the system. The following approach for state management is implemented in the `VariableLagK()` command.

```
# Test running a historical simulation for a longer period, saving states,
# and then restarting at one of those dates
# - Save states on even day
# - Compare the output time series contents from the state save date to the end to
#   make sure startup simulation matches
StartLog(LogFile="Results\Test_VariableLagK_3hr_ScenarioSimulator.TSTool.log")
#
# Read NWSCard input file
ReadNwsCard(InputFile="Data\3HR_INPUT.SQIN",Alias="Inflow")
#
# Route using the same routing parameters used in the mcp3 input deck
# (metric units: Lag(hrs) K(hrs) Q(cms)
# Lag
# K
#    24.0   200.0    12.0  600.00     9.0  1500.0    42.0  3000.0
#    24.0   200.0    12.0  600.00     9.0  1500.0    42.0  3000.0
#
NewTable(TableID="StateTable",Columns="ObjectID,string;DateTime,datetime;StateName;str
ing;StateValue;string")
SetProperty(PropertyName="VariableLagKObjectID",PropertyType=String,PropertyValue="Tes
tSegment")
#=====================================
# Run one time through for the full period and save states every day
VariableLagK(TSID="Inflow",FlowUnits="CMS",LagInterval="Hour",Lag="200,24.0;600,12.0;1
500,9.0;3000,42.0",K="200,24.0;600,12.0;1500,9.0;3000,42.0",InitializeStatesFromTable=
False,StateTableID="StateTable",StateTableObjectIDColumn="ObjectID",StateTableObjectID
="${VariableLagKObjectID}",StateTableDateTimeColumn="DateTime",StateTableNameColumn="S
tateName",StateTableValueColumn="StateValue",StateSaveInterval="Day",NewTSID="TestLoc.
.SQIN.3Hour.routed",Alias="Outflow")
#=============================================
# Now run through one more time starting in the middle of the period,
# and use states from first run
VariableLagK(TSID="Inflow",FlowUnits="CMS",LagInterval="Hour",Lag="200,24.0;600,12.0;1
500,9.0;3000,42.0",K="200,24.0;600,12.0;1500,9.0;3000,42.0",OutputStart="1991-01-01
00",InitializeStatesFromTable=True,StateTableID="StateTable",StateTableObjectIDColumn=
"ObjectID",StateTableObjectID="${VariableLagKObjectID}",StateTableDateTimeColumn="Date
```

```
Time",StateTableNameColumn="StateName",StateTableValueColumn="StateValue",NewTSID="Tes
tLoc..SQIN.3Hour.routed2",Alias="Outflow2")
SelectTimeSeries(TSList=AllMatchingTSID,TSID="Outflow",DeselectAllFirst=True)
SelectTimeSeries(TSList=AllMatchingTSID,TSID="Outflow2",DeselectAllFirst=True)
```

The example illustrates the general approach to managing states:

1. Create a table (or read after initial creation) that will contain the following columns (column names are specified with command parameters):
    a. ObjectID – identifier for the object associated with the states, for example a stream reach (optional if only one object is represented in the table)
    b. DateTime – date/time for saved states
    c. StateName – name of the state being saved (optional if only one state is represented in the table)
    d. StateValue – value of the state, with format that depends on the command that saves states (see below for example)

    States will be written to the table when saving states, and states can later be read from the table. The DateTime and StateValue are required columns. DateTime, or ObjectID + DateTime, or ObjectID + DateTime + StateName are required to lookup states, depending on what is needed to uniquely identify the states.
2. Create/specify a property that contains the object identifier and also the state name to which states will be associated in the table. Both of these are optional if states are matched only by DateTime.
3. When writing states from the command, specify the interval or date/time to write states.
4. When reading states, read states for the date/time at the start of the input time series (default) or the `OutputStart` parameter. Other command parameters may be implemented (such as for the VariableLagK command) to provide default initial states, for cases when a table lookup is not use or is not successful.
5. Write the states table to a persistent format such as a CSV file (TSTool `WriteTableToDelimitedFile()` command) or Excel worksheet (TSTool `WriteTableToExcel()` command).

The above approach can be tailored as appropriate. The `VariableLagK()` command uses a single state variable name for all states, and stores the states in a JSON string, which is inserted into the state table. This minimizes the need to specify command parameters to map state names to separate columns, deal with strings, numbers, etc. column types, and locate separate states. Instead, the JSON represents all the states at a point in time and the software encodes and decodes the JSON. JSON can be handled by multiple software languages.