Group Development Software for Vensim®

David Thompson
Crescent Solutions
1102 Russell Drive
Highland Beach, FL 33487
(509) 624-1018
dt.home@comcast.net

Brian W. Bush
Los Alamos National Laboratory
Los Alamos, NM
(505) 667-6485
bwb@lanl.gov

Abstract: *The development of large software systems using systems dynamics languages such as Vensim® has been hampered by the lack of means to develop modules independently and subsequently link, integrate, or merge the modules. Most modern software languages support such a capability. A software tool to facilitate group development of systems dynamics code and the development conventions to support the process has proved successful in a large code development project. The tool, Conductor, is generally applicable to other projects using Vensim®.*

There are many existing software development paradigms, frameworks and tools. However, systems dynamics languages, such as Vensim®, is significantly different from more conventional software languages such as C, C++, Java, and others, in that the application code is not readily modularized. Thus many of the existing software development approaches are not directly applicable to the systems dynamics programming environment. However, in the context of a large programming project[1] using systems dynamics, it was imperative to develop an approach that enabled modularization of the code and means to link and run it. This process began by defining a set of rules to define the software development process and software development practices (standards) that would ensure the development of quality software.

Standardization Challenges

Important challenges exist in the development of large System Dynamics models. In this document, standardization refers to the aspects of development that must be done consistently, and coordinated with a development team, in order to create coherent software. Standardization includes normalization, modularization, automation tools, and model development practices. These elements of standardization are further understood by what they include.

[1] The referenced project is the Critical Infrastructure Protection Decision Support System, the specific code being the Metropolitan model, developed for the Science and Technology Office of the Department of Homeland Security.

Normalization includes:

- Common semantics/philosophy.
- Common level of detail.
- Common naming practices.
- Common input data accuracy.
- Common diagramming practices (colors, fonts)
- Categorization/taxonomy of variable types.
- Scope (i.e. what one intends to model)
- Metrics

Modularization includes**:**

- Component reuse.
- Pattern reuse.

Automation can include mechanisms to aid in:

- Model merging.
- Managing name spaces.
- Instantiation of reusable components.

Development Practices can include:

- Use of a revision control environment.
- Use of testing mechanisms (test first development, regression testing, automated consistency checks).
- Use of common communication mechanisms for changes in architecture and interfaces.
- Decision making processes such as: consensus, voting, expert opinion, and committee.

Standardization Overview

Many of the abstractions implied by the modularization of System Dynamics models are familiar to the object oriented development world:

- Class (component) specification
- Class instance (component) instantiation
- Namespace issues
- Re-use
- Encapsulation

Unlike standard object oriented development, System Dynamics models require proper coordination of the units among model component interactions. One of the most obvious issues to handle in standardization is namespaces, a means to establish the scope of a variable. In some systems dynamics languages, such as Vensim, variables have global scope. Namespaces are important to normalization, modularization, and automation, as these mechanisms are facilitated by maintaining a local scope for variables. Many of the namespace issues of standard programming languages are handled by compilers and linkers. Object oriented language mechanisms allow for differentiation among specific instance objects by associating the instances with unique references and memory locations.

The Vensim® system dynamics platform does not provide automated namespace handling for differentiation of component instances. However, this technical challenge must be addressed in order to develop more modular code. It may be possible to employ coding specifications to deal with namespaces and reuse of modular components. However, automation can be useful for problems that need to be handled regularly. Large scale System Dynamics development could benefit from automation in:

- Reduced errors due to hand namespace customization.
- Reduced time and effort in reintegrating changed components.
- Increased understanding of code due to use of repeated patterns.
- Reduced quality assurance efforts (due to reduced amount of code).

There are drawbacks to applying modularization and automation to System Dynamics model development. These drawbacks include the need for modelers to have an increased understanding of the complex issues involved in the process. However, in large scale System Dynamics codes, the issues described in this document exist regardless of how one chooses to deal with them. Abstraction, automation and knowledge can be substituted for large time and effort costs over the development cycle. When considering automation, it is important to weigh the cost of developing automation tools to the potential savings while using the automation tools.

Namespaces

How does one avoid clashes in namespaces? These clashes can occur in many ways. If one wishes to re-use components, the issues become even more complicated. In the following discussions, the word “client” refers to a model that would use a modular component in its construction. The word “component” refers to a model that one wishes to use in multiple places within a larger model.

The Problem:

There are seven potential namespace interactions to consider for clashes (in different merging situations), within a large System Dynamics programming project:

1) When merging two sector models, the variables of the merged models may clash with each other.
2) When merging a reusable component into a client model, the variables of reusable components may clash with variables of client models.
3) When merging two *different components* into the same client model, the variables of two different reusable components, used in the same client model, could clash.
4) When merging two sector models that use instances of the same components, the variables of the reused components could clash.
5) When using multiple instances of the *same component* in a single client model, the variables of the two different instances of the same component could clash.
6) When merging multiple instances of a model, all of the variables from each model could clash.

7) When merging segments of a large model with other models (such as related but separate models), the namespaces of the independently developed models could clash.

Potential Solution: Identifiers

Three levels of namespace identifiers may be necessary to resolve these potential clashes:

1) Sector/subsector (in this context, a *sector* refers to a mostly self-contained substantial segment or module of a larger model, while a *subsector* is a logical submodel of the sector, often represented as a separate “view”) prefix identifiers can distinguish the use of variable names between subsector models. This would resolve clashes of type 1 above.

2) Variables in reusable components (in this context, *components* are model fragments that recur in several contexts and are thus candidates for reuse; sometimes called molecules) could have a prefix to identify them with a particular component. This would avoid clashes of types 2 and 3.

3) Sector and subsector client model prefixes could be added to the component prefix. This would avoid variable name clashes of types 4.

4) A component instance identifier could resolve clashes of type 5.

5) A model identifier, added to every variable and subscript in each model, would resolve the clashes of type 6.

6) An identifier that indicates that a model belongs to a set of models would resolve clashes of type 7.

Components

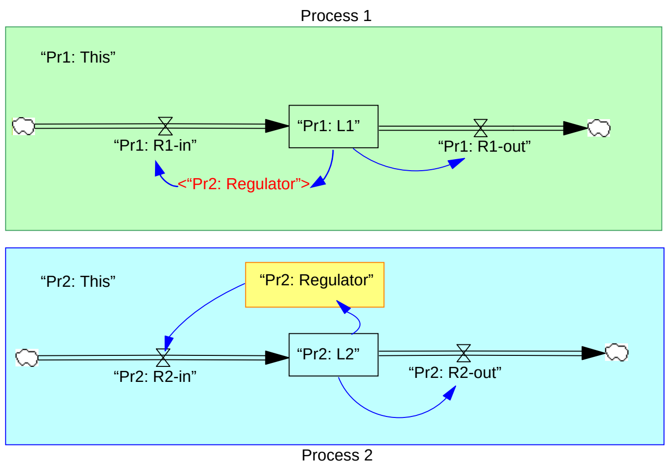
There are several levels of automation to consider for components. In its simplest form, components can be added to models by hand. In an intermediate approach, one could develop models in a package (such as Vensim), and use them as parts that are read into a framework that synthesizes the parts into a single model. The package is used to run and analyze the resultant model. This is probably the more sensible approach when considering the scope of a large Systems Dynamics project.

At the extreme of automation, a framework could be developed for a fully object oriented specification of System Dynamics models. The models could be compiled into Vensim models from the specification language. The Vensim engine could be used to run and analyze models.

To connect a component into a client model, a connection specification must be used. The following must be considered:
1) Components will generally need to receive input values from a client model.
2) The client model will need to receive output from the component model.

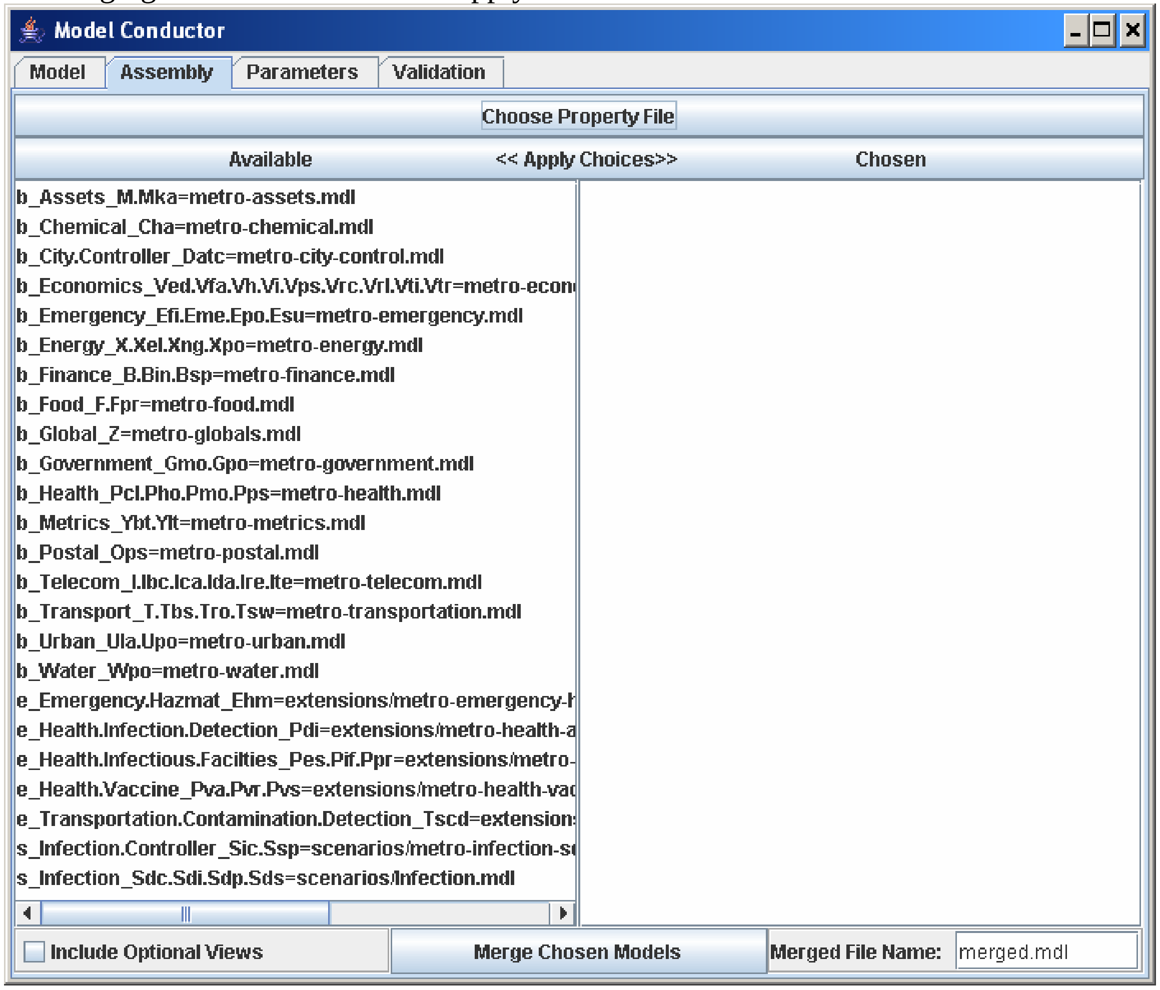
3) The units of variables in a component will not generally match the units required by the client model.
4) A component may require the use of subscript ranges. In this case, the subscript ranges used to specify the component will be different from those used in the client model.

The Merge Concept



The main obstacle to developing separate models for subsequent combination is the management of namespace, e.g., in the example model at left, two simple models, Process_1.mdl and Process_2.mdl depicts the sharing of a single variable "Pr2: Regulator. In Process_1, "Pr2: Regulator" appears as a shadow variable, even though it is not otherwise defined in the file. Normally a variable X in Process 1 will clash with a variable X in Process 2 since there is no differentiating namespace, i.e., there is no information to say that Process 1 X is the same or different from Process 2 X. The convention used in this case is to define a namespace by "<symbol: >This". All variables that appear in the model must have the designated namespace <symbol: > as a prefix. Since Vensim models are text files, they can be parsed to match namespaces and variable names and subsequently written as an integrated model. Variables that appear in the model from another namespace are monitored by the parser and appropriately linked when the source file that defines the namespace is encountered. With this process a large model can be represented as a set of functionally interacting submodels such that defined interfaces exist between the model pieces. In the development process used, each subsystem can be developed independently and subsequently merged into a single file for execution.

Figure 1 shows an example of using the Conductor tool to merge files. File names are given on the left are potentially included in the merge. The user selects the desired files for merging and then clicks the <<Apply Choices>> button.

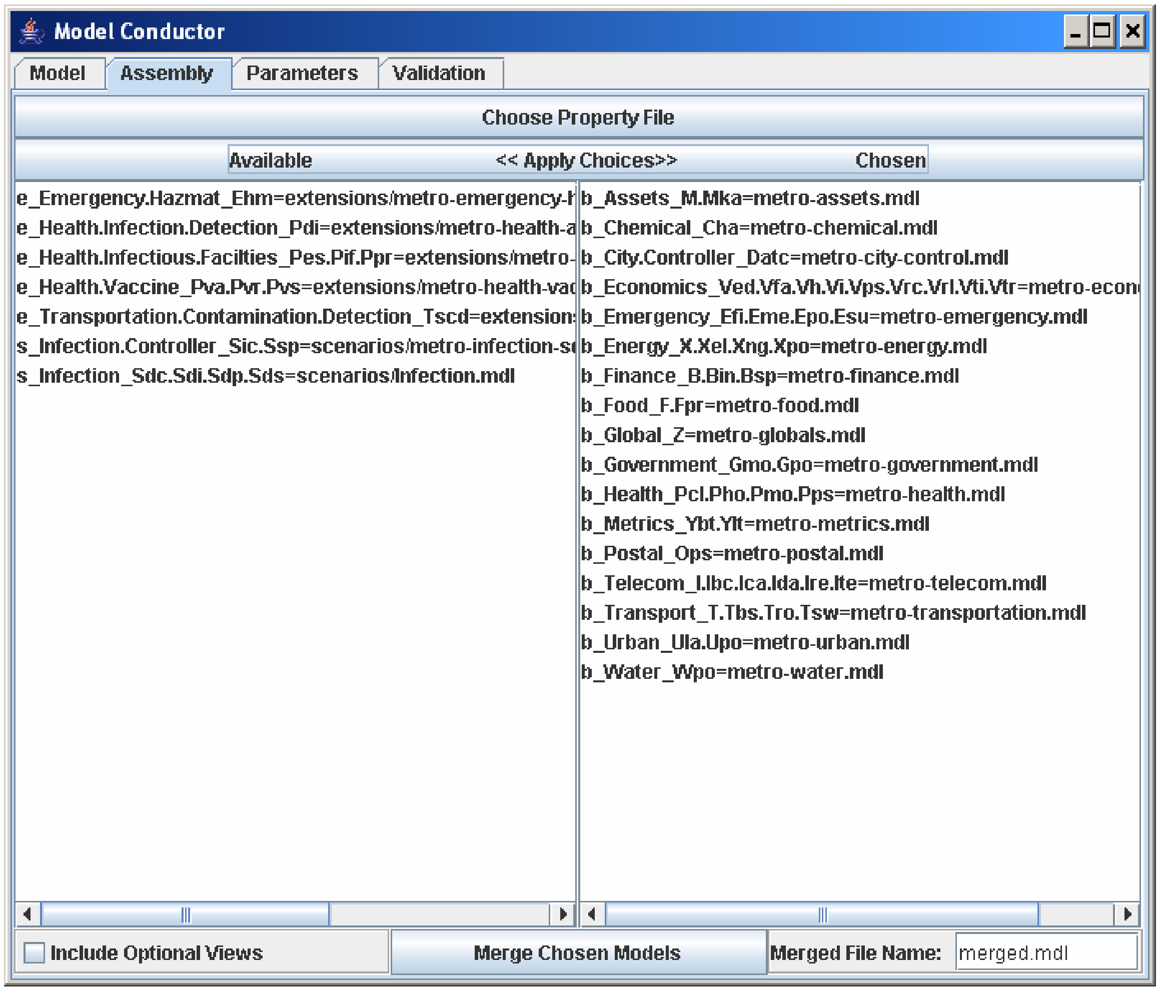


b_Assets_M.Mka=metro-assets.mdl
b_Chemical_Cha=metro-chemical.mdl
b_City.Controller_Datc=metro-city-control.mdl
b_Economics_Ved.Vfa.Vh.Vi.Vps.Vrc.Vrl.Vti.Vtr=metro-econ
b_Emergency_Efi.Eme.Epo.Esu=metro-emergency.mdl
b_Energy_X.Xel.Xng.Xpo=metro-energy.mdl
b_Finance_B.Bin.Bsp=metro-finance.mdl
b_Food_F.Fpr=metro-food.mdl
b_Global_Z=metro-globals.mdl
b_Government_Gmo.Gpo=metro-government.mdl
b_Health_Pcl.Pho.Pmo.Pps=metro-health.mdl
b_Metrics_Ybt.Ylt=metro-metrics.mdl
b_Postal_Ops=metro-postal.mdl
b_Telecom_I.Ibc.Ica.Ida.Ire.Ite=metro-telecom.mdl
b_Transport_T.Tbs.Tro.Tsw=metro-transportation.mdl
b_Urban_Ula.Upo=metro-urban.mdl
b_Water_Wpo=metro-water.mdl
e_Emergency.Hazmat_Ehm=extensions/metro-emergency-h
e_Health.Infection.Detection_Pdi=extensions/metro-health-a
e_Health.Infectious.Facilties_Pes.Pif.Ppr=extensions/metro-
e_Health.Vaccine_Pva.Pvr.Pvs=extensions/metro-health-vac
e_Transportation.Contamination.Detection_Tscd=extension
s_Infection.Controller_Sic.Ssp=scenarios/metro-infection-s
s_Infection_Sdc.Sdi.Sdp.Sds=scenarios/Infection.mdl

merged.mdl

Figure 1. Example of the Conductor user interface.

The selected files are then moved to the right panel, as shown in Figure 2. The user then clicks “Merge Chosen Models” and the files are merged and saved to a user specified file, in this case “merged.mdl”. The merged model can then be loaded into Vensim and be manipulated as any model file. The Conductor has other functions listed in the tabs of the interface, e.g., input parameter manipulation, model examination, and a “validation” function for testing the input code against code development standards. While these functions are useful, they are not central to the concept of merging separate models.



```
e_Emergency.Hazmat_Ehm=extensions/metro-emergency-h
e_Health.Infection.Detection_Pdi=extensions/metro-health-a
e_Health.Infectious.Facilties_Pes.Pif.Ppr=extensions/metro-
e_Health.Vaccine_Pva.Pvr.Pvs=extensions/metro-health-vac
e_Transportation.Contamination.Detection_Tscd=extension
s_Infection.Controller_Sic.Ssp=scenarios/metro-infection-sc
s_Infection_Sdc.Sdi.Sdp.Sds=scenarios/Infection.mdl
```

```
b_Assets_M.Mka=metro-assets.mdl
b_Chemical_Cha=metro-chemical.mdl
b_City.Controller_Datc=metro-city-control.mdl
b_Economics_Ved.Vfa.Vh.Vi.Vps.Vrc.Vrl.Vti.Vtr=metro-econ
b_Emergency_Efi.Eme.Epo.Esu=metro-emergency.mdl
b_Energy_X.Xel.Xng.Xpo=metro-energy.mdl
b_Finance_B.Bin.Bsp=metro-finance.mdl
b_Food_F.Fpr=metro-food.mdl
b_Global_Z=metro-globals.mdl
b_Government_Gmo.Gpo=metro-government.mdl
b_Health_Pcl.Pho.Pmo.Pps=metro-health.mdl
b_Metrics_Ybt.Ylt=metro-metrics.mdl
b_Postal_Ops=metro-postal.mdl
b_Telecom_I.Ibc.Ica.Ida.Ire.Ite=metro-telecom.mdl
b_Transport_T.Tbs.Tro.Tsw=metro-transportation.mdl
b_Urban_Ula.Upo=metro-urban.mdl
b_Water_Wpo=metro-water.mdl
```

Merged File Name: merged.mdl

Organizing Vensim Models

A composite model can be created from multiple sub-models developed independently, provided the developers follow simple conventions. This section describes the conventions used for the project that gave rise to this paper.

Time Unit

Each model should use the same unit for time. Models can be constructed that will operate with the "Units for Time" value set to a common value, such as Minute. The "TIME STEP" can be set to any appropriate value for your particular model (0.1 Minute, 1 Minute, 15 Minutes, 3600 Minutes…). It is convenient to provide conversion factors from all common time units to Minute in a common definition space, e.g., a Global Constants model. These conversion factors can be used anywhere a model equation references time. Use of conversion factors for all equation references to time, will ensure that a model will facilitate proper operation of the model even if the base time reference unit is changed.

Key areas for the application of time conversion factors are listed below. (Note: use the Vensim Check Units facility (Model->Check Units), to ensure that units are properly applied).

1) INTEG: All variables using the INTEG function (or one of the other integration functions), will need unit conversion. Level variables (Stocks) generally employ the INTEG function. Conversion is needed in the INTEG function because integration is with respect to time (recall the factor of dt in an integral).

    For example: If the rate you are integrating is in widgets per Day, use the Global variable "*Z: Time Units Per Day*". Divide your integrand (rate) by "*Z: Time Units Per Day*" to gain the proper conversion.

2) Time: Model variables defined with equations using the Vensim *Time* variable should use the proper conversion from the Global model.

3) Database values: Units being read in from a database may not be consistent with a unit of time required by the model and conversion may be necessary.

Except for the Vensim variable Time, the other Vensim time variables (TIME STEP, FINAL TIME, START TIME), should not be used in equations defining model variables. These Vensim variables can have different values for different simulation runs.

Checking the Units and the Model

After any changes to a model, the developer should check to make sure that the model and the units are correct. To do this (in Vensim):

1) Use the **Check Units** functionality to ensure that all units are consistent.
    - **Check Units** is found in the **Model** menu.
2) Use **Check Model** functionality and address any issues that arise. (Some output variables will cause a complaint that the model is not using the variable. To suppress this complaint, set the variable type to **Supplementary**.)
    - **Check Model** is found in **Model** menu.
    - Note the **Supplementary** check box in the **Equation Editor** window**.**

Names Spaces and Variable Specification

Namespaces are defined by defining the "This" variable by specifying a meaningful prefix, e.g., "MyNamespace: This". In the parent application, meaningful namespaces were chosen by selecting a single capital letter for the model and two lowercase letters for the submodel, e.g., "Tro: This" to define the namespace for a model of Transportation using roads. However, limiting the namespace token to only three characters rapidly became too limiting, so namespace designators were generalized.

Within a namespace, all variable names carry the namespace designator, with a colon (":") and space separating the namespace token from the variable name. The goal here is to keep the clutter introduced by the namespace prefix to a minimum and make the variable readable to non-programmers. Use English phrases so customers can easily read the Vensim diagrams. An example variable would be:

"Tro: Daily Trips" = transportation model, road submodel, defining the total number of daily road trips.

Programmers are encouraged to include a range and a description of each variable in the Vensim equation editor dialog box. The merge program can parse the Vensim text file to create complete variable descriptions (for use in documentation, and by other modelers). Descriptions should begin with a capital letter, and end with a period to facilitate the use of the description in automating documentation. Each variable must belong to a group in the Vensim equation editor dialog box. Groups can be defined in a meaningful way for a given application, but following are the specific groups that were used in the parent application.

Group	**Description**
city-data	Input data specific to a particular city
constant	True constants that do not vary by city or scenario
internal	An internal model variable not generally used outside the subsector
metric	Consequence metric output not generally used by other subsectors
national	Input from the national model
other-sector	Input from another metropolitan sector
output	Output to other metropolitan sectors
output-metric	Output to other metropolitan sectors and consequence metric
scenario-data	Input data specific to a particular scenario
subscript	Subscript types
global-data	The variables in the Global Data model/view that you copy into your model. Please set these variables to this group.

Subscript and Subscript Element Names:

Prefix the name of subscripts with the same letters that define the namespace, however use a space, instead of a colon, to separate the name from the prefix (the colon is a reserved character for the expression of subscripts in Vensim). Use prefixes on each subscript type, and on each element of a subscript type.

- Example Subscript Type: Tro Trip Types
- Example Element of Tro Trip Types: Tro Commercial

Note: If you have multiple subscript types, ensure that all subscript elements in your model have unique names. If you don't ensure unique element names, your new subscript type will show up as a subset of a previous subscript type. The database requires a complete enumeration of elements for each subscript type. This can only be accomplished via Vensim if the elements of each subscript type are globally unique. When integrating models, if elements from subscripts of one model are identical to those in another model, there will be a clash in Vensim.

Naming Views and Graphs

Views containing subsector models should start with the subsector prefix, followed by a colon: for example, “Xel: Electricity” or “Tro: Roads”. If the view is supplementary (a control panel, a collection of shadow variables, etc.), omit the colon and use a dash instead: for example, “X - Control Panel” or “Tro - National Model”.

Graphs should have names starting with the subsector prefix, followed by an underscore: for example, “Xel_Consumption”.

Control View

It can be useful to make a control view for each of your subsector models. A control view is a view containing sliders and graphs that are most important to understanding your model.

1) Create a properly named view such as Ops Control (for the case of Postal).

2) To create a Control View slider, use the Input Output Object (this is a sketch tool located to the left of the “Com” (comment) sketch tool). Click where you want the slider to be placed (with the Input Output Object selected). When the panel comes up, choose Input Slider, then choose a variable that the slider will control.

3) To create a Control View graph, first define a new (custom) graph in the Control Panel (the gauge/clock looking thingy in the upper right). Choose the Graph tab of the Control Panel, then define and name your custom graph. The custom graph’s name should begin with your subsector prefix (Xel_My_Graph). The first time you create a custom graph in your model, hit Save As, in the Graph panel, to save the custom graph definition in a file.

4) Now that you have a custom graph(s), showing variables from your model, return to the Input Output Object. Click where you want the graph to be placed (with the Input Output Object selected). Next, select “Output Custom Graph” and choose your named graph from the dropdown menu.

Documentation

Minimal document should include:
1) A diagram of the conceptual model.
2) An indication of the input that is expected to come from other models.
3) The output (interface) information that will be provided for other models to connect to your model(s) (including any consequence metrics you have identified).

It is possible to generate variable tables from a database. This is an important aid in partial automation of model documentation.

Software Development Process

Like most software projects, the process involves defining the purpose of the software (what is the problem to be solved), identification of requirements, preliminary design of features, functions, and modules, final design, implementation, and quality assurance. Preliminary design of software is a key element in achieving quality code. Given the functional requirements and a statement of the purpose of the code, preliminary design is possible. The design process can be augmented by use case diagrams, swim-lane diagrams, influence diagrams, causal loop diagrams, as well as listings of important inputs, outputs, metrics, and investments. Review of preliminary designs is a high payoff activity to detect any gaps with respect to requirements and assure than software development goals are being met. Preliminary design revision leads to the final design, a reviewed and approved approach that addresses requirements and can be implemented with the allocated resources. A final design is presented in order to move into the actual process of writing code.

Coding standards were prescribed and provided in a document for the development team members. This consisted mainly of usage for the software repository (based on CVS) and on coding standards for Vensim which were developed from previous project experience with Vensim and other published references ([Ford99], [Ster00]). It is these coding standards and the fact that Vensim has a text-based model representation that enabled the development of the Conductor.

Testing is one means to assure that code is functioning properly. It may be desirable in many cases to develop one or more views (a Vensim mechanism to segment code in a model in its own window) consisting of a test panel that executes specific tests on the code. Other testing can be simply performed prior to committing a model to the repository. The Vensim “Model Check” and “Units Check” must be applied and successfully passed. While this level of testing is useful, it does not guarantee an absence of errors.

Conclusion

The development of large system dynamics software codes can be distributed among a team using analogues to modularization to develop code systems that can be combined, i.e., merged to form a single code. The mechanisms to support this methodology are a merge tool, such as Conductor, and code development practices that ensure quality code development such as syntax checks, model checks, unit checks, and code testing.

References

[Ford99] Andrew Ford, Modeling the Environment: An Introduction to Systems Dynamics Models of Environmental Systems, Island Press, Washinton, D.C., 1999.

[Meta01] Marcio de Oliveira Garros, Claudia Maria Lima Werner, Guilherme Horta Travassos, COPPE/UFRJ Computer Science Department, From *Metamodels to Models: Organizing and Reusing Domain Knowledge in System Dynamics Model Development*, Published on the World Wide Web, 2001.

[Pow00] Myrtveit, M., "Object Oriented Extensions To System Dynamics", in *The Proceedings of the 18th International Conference of the System Dynamics Society*, Bergen, Norway, 2000.

[Ster00] John D. Sterman, Business Dynamics: Systems Thinking and Modeling for a Complex World, McGraw-Hill, Inc., Boston, MA., 2000.

[Tho04] David Thompson, D. Powell and B. Bush, "CIP/DSS Metropolitan Team Standardization Guide," Report LA-UR-04-6148 (Los Alamos National Laboratory, 2004).