

Design Notes for GUI Forms

by

Dick Stevens

October 5, 2002

The Processing Graph Method Tool (PGMT) product is being released under the GNU General Public License Version 2, June 1991 and related documentation under the GNU Free Documentation License Version 1.1, March 2000. <http://www.gnu.org/licenses/gpl.html>

I. INTRODUCTION

Included in this document are descriptions of the menus and the various forms associated with the icons. Not included in this draft is a description of the icons and how they may be laid out and connected. We will discuss the following:

- The menus, the items in each menu, and the function of each menu item.
- The information to be presented to the operator in each form.
- The syntax rules of each field in each form, and how the GUI shall enforce the syntax rules.
- The semantics rules of the fields in the forms, and how the GUI shall enforce the semantics rules.
- The mapping between fields in the forms and the syntax elements of the Graph State File (GSF) as defined in the Graph State File Specification, by Kalpana Bharadwaj, January 27, 2000.
- The dynamics of clear, cut, copy, and paste in the forms.

We assume that the workstation has a 2-button mouse. Unless otherwise stated, *click* or *select* refers to the left mouse button. The *pointer* is the icon (usually a bold arrow) that continuously moves with the mouse. The *insertion point* is the point in the screen in a text field where the pointer was at the time of the most recent mouse click. The insertion point may be a vertical bar or an i-beam for text.

This document specifies the Graphic User Interface (GUI) of the Processing Graph Method Tool (PGMT). We refer to the human using the GUI as the *operator*. In the following discussion, a *GUI session* refers to the editing of a GUI graph. While the operator is editing the GUI graph, the internal data structures in the GUI memory are called the *parse tree*. The operator may elect to *save* the GUI graph in a file, called the *Graph State File*, or *GSF*. After saving the GUI graph in a GSF, the operator may start a new GUI session and open the previously saved GSF. The state of the parse tree and the information presented to the operator are identical to what they were when the GUI graph was saved.

We will speak of "user-defined data types" or "user-defined classes". In this context, the *user* is the programmer who defines the data type or class. The user may or may not be the same person as the operator. This GUI specification does not support the definition of user-defined data types; rather the user defines classes in the target language (C++). The user is responsible for testing any classes so defined and for applying MTOOL to those classes.

Section II describes the menus. Section III gives the methods by which the operator can open the various forms. Section IV provides some general information about the forms. Section V describes tables that appear in the various forms. Section VI describes the Forms. Appendix A gives the system-defined read-only forms. Appendix B gives an example of a graph, including its graph state file, the layout of its icons and arc connections, its forms, and resulting translated C++ code.

II. MENUS

There are six menus in the menu bar, as follows:

II.A File

The File menu has the following items:

- **New** Start a session to create a new GUI graph. The GUI graph is initially blank.
- **Open** Open an existing GSF. The GUI shall allow the operator to select an existing GSF via a file browser. When opening an existing GSF, the GUI shall verify that the stem of the filename is the same as the Prototype Name in the Prototype Form. If they are different, the GUI shall inform the operator and query for a new Prototype Name, offering to change it to the stem of the filename.
- **Close** Close the current GSF. The operator shall have closed all forms. If any changes have been made to the GUI graph since it was last saved, the GUI shall ask the operator whether to save it before exiting.
- **Save** Save the current GUI Graph in a GSF. The file name of the GSF shall be the Graph Class Name (specified by the operator in the Prototype Name in the Graph Prototype Form, as described in §VI) appended with extension ".gsf" .
- **Save As** Save the current Graph in a GSF in an operator-specified directory via a file browser. The name of the GSF shall be as described in the "Save" menu item above.
- **Recent Files** Open a sub-menu of recently edited Graph State Files. The operator may select one of these to be opened for editing.
- **Exit** Exit the GUI. The operator shall have closed the Graph State File.

II.B Prototypes

The Prototypes Menu has the following items:

- **New Ord Tran:** Open a new Prototype Form for an Ordinary Transition.
- **New Queue:** Open a new Prototype Form for a non-standard queue.
- **New GVar :** Open a new Prototype Form for a non-standard graph variable.
- **Operator-defined:** Open a sub-menu of existing operator-defined Transition and Place Prototype Forms. The operator may select a Prototype Form from this menu to open and edit.
- **System-defined:** Open a sub-menu of system-defined Prototype Forms: Pack, Unpack, Queue, and GVar. The operator may select one of these Prototype Forms to open read-only. The system-defined Queue and GVar Prototype Forms specify a single input port and a single output port (i.e., the port families have height 0).

II.C Exterior

The Exterior Menu has forms related to the GUI Graph being edited. The operator may select any of these forms to open and edit:

- Prototype
- Port Association
- Banner
- Type List
- Included Graph List

II.D Translation

The Translation Menu has the following items:

- **Validate Graph** Validate the graph according to the semantics rules specified below in this document. Report errors to the operator.
- **Output C++** Validate the graph as described in the preceding bullet. If there are no errors, generate C++ code for the current GUI graph according to the Translator Specification, by Dick Stevens, January 19, 2000.

II.E Action

The Action Menu has the following items:

- **Undo** Undo the last operator action. The GUI shall maintain an edit history of changes in the parse tree for the GUI Graph since starting the current session. Repeated "Undo" operations shall undo changes in reverse chronological order, undoing the most recent change first.
- **Clear** Delete selected graphic or textual items.
- **Delete** Delete selected item(s).
- **Cut** Delete selected items and place them in the clipboard.
- **Copy** Place selected items in the clipboard without deleting them.
- **Paste** Paste the contents of the clipboard at the insertion point. If the clipboard contains graphic information, then the pointer must be in a graphic region of the screen (i.e., not in a form). Similarly, if the clipboard contains text information from field(s) of a form, the pointer must be in a text field of a form that is appropriate for the text in the clipboard.
- **Print** Print all or a selected subset of the graphic view of the graph, its forms, and the forms of its icons. Specifically, the operator may print a single form to be printed, all the forms relating to a selected icon, the graphic layout of the graph, or all the forms of all icons in the graph. This variety of options may be supported in the GUI by a sub-menu of the "Print" in the action menu.
- **Show Tool Bar** Toggle between displaying and hiding the Tool Bar. The tool bar contains buttons for use in creating and selecting icons in the graphic screen, as well as other operator actions relating to the layout of the GUI graph on the screen. Items in the Tool Bar are also available via the Nodes Menu.

II.F Nodes

The Nodes Menu contains the following items, which are also displayed in the Tool Bar. These items are described in another document by David Kaplan (document TBS):

- Select
- Transition
- Place
- Included Graph
- Arc
- Arc Bends

III. OPENING FORMS

All forms associated with the graph are opened by selection from the Exterior Menu as described in §II.C above. In this section we describe how to open the various forms for each icon and for arcs. There are three kinds of icons: Transition icons, Place icons, and Included Graph icons. For each icon, there are three kinds of forms, each of which may be opened by pointing at the icon and pressing the right mouse button. This opens a pop-up menu from which the operator can select one of three forms associated with the icon. The three forms are the Prototype Form, the Call Form, and the Icon Arc Form. Finally, there is an Arc Form for each Arc, which specifies how the Ports of two respective nodes should be connected.

III.A Icon Prototype Form

Every icon has a Prototype Form. It is possible for several icons to have the same Prototype Form. The Prototype Form for a given icon may be opened read-only from the same pop-up menu described above. Some Prototype Forms (i.e., for Pack and Unpack Transitions and for standard Queues and Graph Variables) are read-only, and the operator cannot edit them. To open and edit an Ordinary Transition Prototype Form or non-standard Place Prototype Form, the operator shall select the desired Prototype Form from the Prototype Menu on the menu bar. To edit the Prototype Form for an Included Graph, the operator shall use the GUI to open the GSF of the underlying graph and edit the Graph Prototype Form for the underlying graph.

III.B Icon Call Form

Every icon has a unique Call Form. To open and edit the Call Form for a given icon, the operator points at the icon, presses the right mouse button, and selects the Call Form in the pop-up menu.

III.C Icon Arc Form

The Arc Form for a given icon displays some of the arc information for every arc connected to the icon. The Arc Form for a given icon may be opened read-only from the same pop-up menu described above for opening the icon's Call Form.

III.D Arc Form of an Arc

To edit an arc form for a given arc, the operator has two options for opening the desired arc form:

- The operator may point at the desired arc and click the right mouse button.
- The operator may select the desired arc from the icon arc form of one of the icons to which the arc is connected and click the right mouse button.

IV. General Notes

IV.A Editing Forms

To maintain coherence of the parse tree, while the operator is editing a form, the GUI shall block the operator from doing anything that either results in a change of the parse tree or has the potential to do so. In particular, the operator may edit no more than one form at a time. While editing a given form, the

operator may view any other form, but may not edit it. Further, while editing a given form, the operator may not add icons or arcs and may not move any icons or in any other way cause the graph display to change. To complete editing a form in order to perform another action, the operator shall finish editing the current form by clicking on one of the three buttons "OK", "Apply", or "Cancel" in the current form.

If the operator is editing a form and opens, closes, or saves a file, or creates a new file, the GUI shall first display a message asking the operator to close the form.

Some information that the operator enters in one form will also appear in other forms. When the operator edits one form and clicks either "Apply" or "OK", the GUI shall, whenever possible, update all opened forms containing the changed information. If this update is not possible (because the information is either incomplete or inconsistent), the GUI shall so inform the operator.

The GUI shall support the actions "Clear", "Cut", "Copy", and "Paste" for text in the forms. These may apply to text in a field, to any set of fields. "Paste" shall occur at the insertion point, provided that the format of the contents of the clip board are consistent with the insertion point, and shall replace whatever text or fields are selected, if any. It shall be possible to copy in one form and paste in another.

In each field, unless otherwise stated, the operator shall enter text. The syntax rules for each field will specify one of the following forms:

- Variable Name: The normal rules for forming a variable name in most modern languages. The first character is alpha, and each subsequent character is either alphanumeric or an underscore. No embedded blanks.
- Type Name: A name of either a language-defined type (int, float, signed int, ...) or a user-defined type.
- Number: A literal number.
- Expression: An expression according to standard rules for most modern languages. Operands may be literal numbers, variable names, indexed arrays, and function calls.
- Full Path: The full path of a file. In a Unix file system, the directory separator is a slash '/'. The GUI shall provide a file browser to help the operator specify a Full Path.
- Text: No specified syntax for Text. The GUI need not parse Text.

IV.B Validating the Graph

At any time that no form is being edited, the operator may select "Validate Graph" in the Translation Menu. In response, the GUI shall check all forms of the current GUI graph against the semantics rules and report all errors to the operator.

IV.C Translating the Graph

At any time that no form is being edited, the operator may select "Output C++" in the Translation Menu. In response, the GUI shall first validate the graph. If there are no errors, then the translator shall generate the C++ source code as specified by the Translator Specification. The translated code shall be written to a file whose name is the Graph Name appended with extension ".h".

IV.D Saving the Graph

The operator may select "Save" or "Save As" in the File Menu to write a GSF containing all the information of the parse tree in the format specified by the GSF Spec. The GSF name shall be the Graph Name appended with extension ".gsf".

IV.E Family Height

A complete description of Families is beyond the scope of this document. We note at this point that whenever a family height is 0 (indicated either directly or by an empty family tree), the corresponding family is, by definition, a *leaf*. By that we mean that it is an object of the indicated base type and has no family tree.

IV.F Graph State File Specification

We assume that the reader is familiar with the Graph State File Specification (GSF Spec), by Kalpana Bharadwaj. The Bachus-Nauer Form (BNF) in the GSF Spec defines the parse tree maintained by the GUI. In this document we will refer to the Extended BNF (EBNF) in the GSF Spec.

IV.G Notes about Each Form

Following each form we give two kinds of rules for the operator to follow and for the GUI to enforce when filling out the forms:

- Syntax Rules: The GUI shall enforce the syntax rules whenever the operator clicks either the "Apply" or "OK" button in a form. Unless otherwise stated, a blank field is acceptable.
- Semantics Rules: The GUI shall enforce the semantics rules whenever the operator selects either "Validate Graph" or "Output C++" in the Translation Menu. Some of the semantics rules cannot be enforced by the GUI, because there is insufficient information available. An example of this is a dependence on the value of an expression with operands whose values are not known to the GUI.


We also give the mapping between the fields and the EBNF, as specified in the GSF Spec.

V. TABLES

Before describing each of the forms we describe expandable tables and fixed tables, which appear in some of the forms.

Many forms have expandable tables. Here are the general rules for an expandable table:

- The Table may have a title that is centered and underscored.
- There are column headings, one column heading above the table for each column in the table.
- Initially the table is empty (i.e., no rows).

- There are two buttons at the left of the table: one to add a new row and one to delete an existing row. These buttons are depicted by the following symbol: 
- If a field in the table is selected and the operator clicks the add button, the new row follows the row containing the selected field. All existing rows below the inserted row are moved down.
- If no field in the table is selected and the operator clicks the add button, the new row is at the top of the table, and all existing rows are moved down.
- If the table is previously empty and the operator clicks the add button, the new row is the only row in the table.
- If a field in the table is selected and the operator clicks the delete button, the row containing the selected field is deleted.

An empty expandable table looks like this:

<u>Title</u>
Column 1 heading Column 2 heading

An expandable table with two rows looks like this:

<u>Title</u>
Column 1 heading Column 2 heading

Some forms have tables with a fixed number of rows. As in expandable tables, there are column headings. However, there is no button for adding or deleting a row.

In general, a table may have zero or more rows. In Section VI, all tables in the Forms we show will have two rows.

Mapping to EBNF:

- The information in a table provides information about a list of similar things. In most cases, this list will be mapped to a list of similar things in the EBNF. Unless otherwise stated, the order in the table is preserved in the list in the EBNF. Specifically, the top row in the table maps to the first item in the list.

Copy & Paste:

- The operator may select part or all of the text in a field, part or all of the fields in a row in a table, or an entire table to clear, cut, or copy.
- To paste a row or entire table, the operator shall set the insertion point at an appropriate cell in the row or table and select the desired action in the Action Menu. If the contents of the clipboard are consistent with the indicated row or table, the existing row or table is replaced.

V.A Family Tree

A Family Tree is specified by a table, in most cases an expandable table:

<u>Family Tree</u>		
Index	Lower Bound	Upper Bound

Syntax Rules:

- Each Index is a Variable Name.
- Each Lower and Upper Bound is an expression.
- Each operand in the lower and upper bound expressions will be specified in the description of each Family Tree. In general, each operand of an expression shall be a literal, a GIP, or a previously occurring Index in the Family Tree, i.e., an index in a row above the expression. In each case, we will specify the rules for the operands of the expression. Intuitively, one should imagine that a family tree specifies a nested loop, where the first row gives the loop variable and bounds of the outermost loop.

Semantics Rules:

- The indices shall be unique within the Family Tree.
- (Not Enforceable by the GUI) In each row, the Lower Bound and Upper Bound expressions, when evaluated, give the inclusive limiting values of the index.

Mapping to EBNF:

- Each row of a family tree defines a Range. The Index is the Name, the Lower Bound is the first Expr, and the Upper Bound is the second Expr.
- The Family Tree is a sequence of Ranges, the top row being the first Range in the sequence.

Not every Family Tree table is expandable. As in any table, when a Family Tree table is fixed, the buttons to add and delete shall be omitted.

V.B Family Tree in a Table

It is possible for a Family Tree to occupy a cell in a table. If so, then every cell in the column shall be occupied by a Family Tree, and all shall be expandable or all shall be fixed. We will indicate this in the following way for expandable Family Trees:

<u>Title</u>	
Column 1 heading	Column 2 heading
	<input type="checkbox"/> <i>Exp Fam Tree</i>
	<input type="checkbox"/> <i>Exp Fam Tree</i>

If the Family Trees are fixed, we write *Fix Family Tree*. If the operator double-clicks the button in one of the cells, a Family Tree Form opens for the operator to edit.

VI. FORMS

Every form has a title that identifies what kind of form it is. The title is centered and bold.

Although we do not depict them in this document, every form shall also have a three buttons, titled "Apply", "OK", and "Cancel". When the operator clicks on one of the buttons, the following respective actions shall occur:

- Apply: Perform the syntax check of every field in the form and report all errors to the operator. If there are no errors, update the parse tree in memory to incorporate changes made in the form.
- OK: Perform the actions described for "Apply" and then close the form.
- Cancel: Close the form without checking against the syntax rules and without making any changes to the parse tree in memory.

Note that no semantics checks are made in response to clicking any of these three buttons. The semantics checks are made when the operator selects "Validate Graph" in the Translation Menu. It may not be possible for the GUI to enforce some of the semantics rules because of insufficient information. Unless otherwise stated, all rules for the operator shall be enforced by the GUI. Whether enforced by the GUI or not, all rules are enforced either during compilation or during graph construction.

VI.A Graph Banner Form

Graph Banner	
File Name:	<input type="text"/>
Author:	<input type="text"/>
Version:	<input type="text"/>
Purpose:	<input type="text"/>
Main Graph:	<input checked="" type="radio"/>
Included Graph:	<input type="radio"/>

Syntax Rules:

- The File Name shall include the full path.
- The Author, Version, and Purpose are all Text.
- Exactly one of the radio buttons for Main Graph and Included Graph shall be selectable. The default is Main Graph.

Semantics Rules:

- The file name shall be the Graph Prototype Name (taken from the Graph Prototype Form) with extension ".gsf". We suggest that the GUI provide the file name automatically.

Mapping to EBNF:

The contents of the Graph Banner Form shall be mapped to respective terminals in the non-terminal Banner of the EBNF. The selection of Main Graph or Included Graph shall determine the value of the terminal number of the graphtype in the Banner.

VI.B Included Graph List

The Included Graph List has an expandable table:

Included Graph List	
Included Graph Prototype Name	Full Path of Graph State File

Syntax Rules:

- In each row of the table, the Included Graph Prototype Name shall be a Variable Name.
- The Full Path of Graph State File shall be selected via a file browser.
- Every included graph called in the graph shall be listed in this table exactly once.

Semantics Rules:

Mapping to EBNF:

The contents of the Included Graph List shall be mapped to the respective terminals in the non-terminal InclForm in the Exterior. Each row maps to a pair in the sequence of the InclForm:

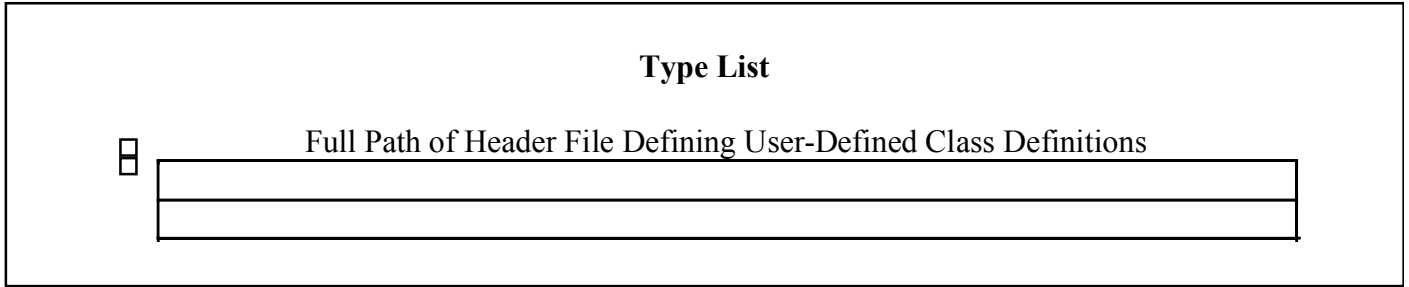
- The Included Graph Prototype Name maps to Name.
- The Full Path of Graph State File maps to PathName.

VI.C Function List

At present, the Function List is not necessary. Its specification is TBS. The GUI shall not implement the Function List. We mention it here, because the GSF Spec stipulates the non-terminal Function in the Exterior.

VI.D Type List

The Type List has an expandable table:



Syntax Rules:

- Each entry shall be composed of an optional Type Variable and a Full Path separated by '@'. The Full Path of each header file defining Classes shall be selected via a file browser.

Semantics Rules:

- Every Full Path shall identify a header file containing user-defined classes. Every Type Variable used throughout the graph shall be listed in this table at most once.
- (Not enforceable by the GUI) For each instantiation of a class template, the operator shall define an explicit type (i.e., without template arguments) via typedef in the source defining the user-defined class. Each Type Variable used throughout the graph shall be defined in one of the Header Files. (Note: if a Type Variable is not listed explicitly in the Type List, the GUI does not have enough information to recognize its definition in the Header Files.)
- The operator shall ensure that each operator-defined class is processed by MTOOL as specified in the MTOOL User Manual, by Michal Iglewski and Roger Hillson.

Mapping to EBNF:

The contents of the Type List shall be mapped to the respective terminals in the non-terminal TypeForm in the Exterior. Each row maps to a pair in the sequence of the InclForm:

- The User-Defined Class Name maps to Name.
- The Full Path of header file defining the Class maps to PathName.

VI.E Graph Port Associations Form

The Graph Port Associations Form identifies each graph port family with a family of ports of an icon in the graph.

Graph Port Associations

<u>Graph Input Ports</u>		
Graph Input Port Family Name	Icon Family Name	Icon Input Port Family Name

<u>Graph Output Ports</u>		
Graph Output Port Family Name	Icon Family Name	Icon Output Port Family Name

Syntax Rules:

- All entries in the two tables of this form are Variable Names.

Semantics Rules:

- In each table, the GUI shall automatically fill in the left-most column with the Graph Port Family Names of the graph, as listed in the respective expandable table of the Graph Prototype Form. Thus the left-most column is read-only.
- For each Graph Port Family the operator shall enter the Icon Family Name and Port Family Name to be associated with the Graph Port Family.
- The Icon Family Name shall be the Family Name of some icon in the graph.
- The Port Family Name shall be the Family Name of some port in the Icon's Prototype Form.
- No Icon Port Family may be associated with more than one Graph Port Family.
- The category of each Graph Port and the associated Icon Port shall have the same category (as specified in the respective prototype forms (see §VI.F below).
- The *direction* of the graph port and the icon port shall be the same (i.e., graph input ports may be associated only with icon input ports, and the same for output ports).
- The mode (i.e., token height and base type) of the Graph Port and the associated Icon Port shall also be the same (height to be enforced by the GUI only if in both cases it is defined by a constant).
- (Not enforceable by the GUI) The family tree of each graph port family (specified in the Graph Prototype Form) shall match the family tree of the associated icon port family (specified in the icon's Prototype Form).

Mapping to EBNF:

The content of the Graph Port Associations form map to the non-terminal PortAssoc in the Exterior. Each row in the Graph Input (Output) Ports table maps to one of the associations in the InAssoc (OutAssoc). In each row:

- The Graph Port Family Name maps to the first name in the association.
- The Icon Family Name maps to the label.

- The Icon Port Family Name maps to the second name in the association.

We suggest that the GUI allow the operator to select the Icon Family Name from a menu of existing icon names in the graph. Further, we suggest that the GUI allow the operator to select the Port Family Name from a menu of existing ports in the icon.

VI.F Prototype Form

Prototype Form

Prototype Name: Entity Type:

Formal Type Arguments (Formal Name)

Formal Mode Arguments

Token Height (Formal Name) Base Type (Formal Name)

Formal Graph Instantiation Parameters (GIPs)

Name Height (Number) Base Type (Actual)

Input Ports

Family Name Category Token Ht Base Type Family Tree

				<input type="checkbox"/> <i>Exp Fam Tree</i>
				<input type="checkbox"/> <i>Exp Fam Tree</i>

Output Ports

Family Name Category Token Ht Base Type Family Tree

				<input type="checkbox"/> <i>Exp Fam Tree</i>
				<input type="checkbox"/> <i>Exp Fam Tree</i>

Open Body

The Prototype Form captures the interface information (i.e., the formal parameters) needed to create an instance of a Graph, Included Graph, Transition, or Place. The Prototype Form depicted above includes all the information for all kinds of Prototype Form. In some cases, some of the information is not relevant and should be disabled. Where this is the case, we will so indicate. Specifically, for a Place, the Prototype Form is significantly reduced from the above depiction. We will give a depiction of the Prototype Form for a Place and discuss the differences in §VI.F.1 below.

The Prototype Forms for special transitions Pack and Unpack are special cases of this with system-supplied entries in their fields. These Prototype Forms are read-only and are given in Appendix A. Also included in Appendix A are the system-supplied read-only Prototype Forms for standard queues and

graph variables. A standard queue or graph variable has a single input port and a single output port (A *single* port is equivalent to a port family with height 0).

To open the Prototype Form for a given Icon via its pop-up menu, the following are required:

- The Prototype Name shall be entered in the Icon's Call Form, and
- The Prototype Form with that Prototype Name shall exist.

When opened in this way, the Prototype Form is read-only. To edit an icon's Prototype Form, the operator shall open the form via the Prototypes Menu (see §II.B). To open the Prototype Form for the current GUI Graph, the operator shall select "Prototype" from the Exterior Menu (see §II.C).

Syntax Rules:

- The Prototype Name shall be a Variable Name.
- The Entity Type shall be "graph", "transition", "queue", "gvar", or "inclgraph", automatically set by the GUI according to the following rules:
 - If the Prototype Form is the GUI Graph Prototype Form, opened via the Exterior Menu, the Entity Type shall be "graph". The Entity Type "graph" indicates that the Prototype Form is associated with the current GUI Graph. This not to be confused with the Banner Form (§VI.A), where the operator may choose between "Main Graph" and "Included Graph". The choice in the Banner Form indicates that the GUI Graph is to be instantiated either as a Main Graph (i.e., by the Command Program) or as an Included Graph (i.e., in another graph).
 - If the Prototype Form is associated with one or more Icons in the GUI Graph, the Entity Type shall reflect the kind of the associated Icons:
 - If the Prototype Form is newly created for an Ord Tran, Queue, or GVar by selection from the Prototypes Menu (§II.B), then the Entity Type is "transition", "queue", or "gvar", respectively.
 - If the Prototype Form is subsequently opened, the Entity Type is taken from the Parse Tree (first element in the non-terminal NodeProto in the EBNF).
 - If the Prototype Form is opened by selection from an Included Graph Icon's pop-up Menu (see §III), then the Entity Type is not derived from any element of the parse tree. In this case, the Entity Type is "inclgraph". The Included Graph Icon's Prototype Form is read-only and, except for the Entity Type field, is identical to the Prototype Form of the underlying GUI Graph. This underlying GUI Graph is in the respective GSF whose full path is found in the Included Graph List (§VI.B).
- Each Formal Type Argument shall be a Variable Name.
- In each Formal Mode Argument, the Token Height and Base Type shall be Variable Names.
- The Name of each Formal GIP shall be a Variable Name. The Height shall be an integer 0 or greater. The Base Type shall be a Variable Name.
- In the tables of Input Ports and Output Ports, for each Port,
 - The Family Name shall be a Variable Name.
 - The Category shall be either "transition" or "place".
 - The Token Ht shall be an expression.
 - The Base Type shall be a Type Name.
- The operator may enter any Text into the Transition Statement.
- If the operator clicks the button labeled "Open Body", the GUI shall open a new window that displays the respective transition statement or graph according to the following:

- If the Entity Type is "graph" the button shall be disabled, because the graph to be displayed is the current GUI Graph being edited.
- If the Entity Type is "transition", the window shall be a window containing Text. This shall be editable or read-only as the Prototype Form is editable or read-only.
- If the Entity Type is "queue", or "gvar", the button shall be disabled, because a place has no body.
- If the Entity Type is "inclgraph", the window shall display the graphic layout of the underlying GUI graph whose GSF is in the respective full path found in the Included Graph List (§VI.B). This graphic layout window is read-only. To edit the underlying GSF, the operator shall open the GSF in a separate GUI session.

Semantics Rules:

- The Prototype Name shall be unique among all Prototype Names in the GUI Graph.
- The Entity Type is described above in the Syntax Rules.
- The Formal Type Arguments shall be unique in the Prototype Form and in the GUI Graph Prototype Form.
- In each Formal Mode Argument, the Token Height and Base Type shall be unique in the Prototype Form and in the GUI Graph Prototype Form.
- In the table of Formal GIPs,
 - The Names shall be unique in the Prototype Form.
 - Each Base Type shall be either a language-defined type (int, float, ...) or a user-defined type that is defined in one of the files in the Type List (see §VI.D).
 - Each GIP is a variable that represents a token of the specified Height and Base Type. If the specified Height is 0, then the variable is a simple variable with the specified Base Type.
- In each of the tables of Input Ports and Output Ports,
 - The Family Names shall be unique among Variable Names in the Prototype Form.
 - If the Prototype Entity Type is "transition", then each Category of each Port shall be "transition". If the Prototype Entity Type is "queue" or "gvar", then each Category of each Port shall be "place". If the Prototype Entity Type is "graph", then each Category may be either "transition" or "place", according to operator choice. If the Prototype Entity is "inclgraph", then the Prototype Form is read-only. NOTE: If in the Banner Form of the graph being edited, Main Graph is selected, then all port categories of the GUI Graph Prototype Form must be "place".
 - Each Base Type shall be either a language-defined type or a user-defined type that is defined in one of the files in the Type List (see §VI.D).
- In each Family Tree,
 - Each index shall be unique among the indices in its Family Tree and among all Variable Names in the Prototype Form. It is not necessary that indices be unique across different Family Trees in the Prototype Form.
 - In each expression specifying a Lower or Upper Bound, each operand shall be a literal number, a Formal GIP in the same Prototype, or an index in a row above the expression in the same Family Tree.
 - (Not enforceable by the GUI) If the Prototype Entity Type is "transition", then the Height of the Family Tree (i.e., the number of rows in the Family Tree table) shall not exceed the respective Token Ht.
- (Not enforceable by the GUI) If the Entity Type is "transition", then the Body is the Transition Statement shall be the body of a function which, when called, specifies the function of the

Transition. The operator may assume that the Family Name of each Input Port and of each Output Port has been declared to be a variable that represents a token with the specified Height and Base Type. If the specified Height is 0, then the variable is a simple variable with the specified Base Type. Further, when the Transition Statement is called, the Name of each Input Port is initialized with the value of the token read from the respective Transition Input Port.

Mapping to EBNF:

- Except as noted, the information in the Prototype Form maps into the non-terminal Prototype.
- The Prototype Name is mapped to the Name in the non-terminal ProtName.
- If the Entity Type is "graph", then the Entity type does not map directly into the EBNF. All other fields of the Prototype Form are mapped to the non-terminal Exterior. If the Entity Type is "transition", "queue", or "gvar", then the Entity Type maps to the respective keyword in the non-terminal NodeProto. If the Entity Type is "inclgraph", the Prototype form is read-only and reflects the Prototype Form of the GUI graph in the respective file in the Included Graph List (§VI.B). All fields are mapped respectively, as described above in the Syntax Rules.
- Each Formal Type Argument maps to a Template in the non-terminal ProtName.
- Each Formal Mode has two parts: Token Height and Base Type. These map respectively to the first and second Name in a FormalArg in the FamName.
- Each Formal GIP has a Name, Height, and Base Type. The name maps to the Name in the Gips, and the Height and Base Type map respectively to the TokHt and BaseType in the Mode of the Gips.
- The two tables titled Input Ports and Output Ports map respectively to Inports and Outports, which have similar structure in the EBNF. In each table, each Port has several attributes, identified respectively by its column heading. Each Port maps as follows to an item in the respective Inports or Outports:
 - Family Name maps to Name.
 - Category maps to Cat.
 - Token Ht and Base Type map respectively to TokHt and BaseType in Mode.
 - The Family Tree maps as described above to the Range list.
- The Transition Statement maps to TrStmt.

VI.F.1 Place Prototype Form

Place Prototype

Prototype Name: Entity Type:

Formal Mode Argument

Token Height (Formal Name)	Base Type (Formal Name)
height	base_type

Formal Graph Instantiation Parameters (GIPs)

Name	Height (Number)	Base Type (Actual)

Input Port

Family Name	Category	Token Ht	Base Type	Family Tree
INPUT	place	height	base_type	<input type="checkbox"/> <i>Exp Fam Tree</i>

Output Port

Family Name	Category	Token Ht	Base Type	Family Tree
OUTPUT	place	height	base_type	<input type="checkbox"/> <i>Exp Fam Tree</i>

The Prototype Forms of non-standard queues and graph variables are operator-defined. The operator may open and define a Prototype Form for a non-standard queue or graph variable by selecting New Queue or New GVar from the Prototype Menu (see §II.B). The Place Prototype Form depicted here is a subset of the Prototype form depicted in §VI.F above. Except as noted, the Place Prototype Form conforms to the syntax and semantics rules and to the Mapping to EBNF for Prototype Forms (see §VI).

Syntax Rules:

- The Entity Type shall be either "queue" or "gvar".
- In each of the tables of Input Ports and Output Ports, there is exactly one Port, and the tables are fixed with one row each.
 - The Family Names shall be "INPUT" and "OUTPUT", as shown.
 - The Category shall be "place", as shown.
 - For the Token Ht and Base Type, the GUI shall repeat the height and base_type that appear in the Formal Mode Argument. These specify the mode of the tokens stored in the place.
 - The operator may edit either or both of the Family Trees in the Input and Output Port tables.

Semantics Rules:

- The Prototype Name shall be unique among all Prototype Names in the GUI Graph. In particular, the Prototype Name shall not be Pack, Unpack, Queue, or GVar.

VI.G Call Form

For each Icon in the GUI Graph, there shall be a unique Call Form. The Call Form identifies the Prototype Form for that Icon and specifies the bindings of the formal arguments specified in the Prototype Form. The Call Form for an Icon is opened for editing via the pop-up menu.

Call Form

Icon Family Name:
 Prototype Name:

Icon Family Tree

Index	Lower Bound	Upper Bound
<input type="text"/>	<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>	<input type="text"/>

Actual Type Arguments

Formal Type	Base Type
<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>

Actual Mode Arguments

Formal Height	Actual Height	Formal Base Type	Actual Base Type
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>

Actual GIP Bindings

Formal GIP	Family Tree	Value
<input type="text"/>	<i>Fix Fam Tree</i>	<input type="text"/>
<input type="text"/>	<i>Fix Fam Tree</i>	<input type="text"/>

Initial Value

Queue Token Index: Lower Bound: Upper Bound:

Index	Lower Bound	Upper Bound
<input type="text"/>	<input type="text"/>	<input type="text"/>
<input type="text"/>	<input type="text"/>	<input type="text"/>

Value:

This depicts a general Call Form, which applies to all kinds of Icons. In the rules that follow, the GUI shall adapt the Call Form to match the interface specified in the Icon's Prototype Form. Therefore, before an Icon's Call Form can be opened, the Prototype Form for the Icon must be identified. If the Icon's Prototype Form has not been identified,

- The GUI shall query the operator for the Prototype Name of an existing Prototype Form in the graph.
 - If the Icon is a Place Icon, the Prototype Name shall be "Queue", "GVar" or the Prototype Name of an operator-defined non-standard queue or graph variable.

- If the Icon is a Transition Icon, the Prototype Name shall be "Pack", "Unpack", or the Prototype Name of an operator-defined ordinary transition.
- If the Icon is an Included Graph Icon, the Prototype Name shall occur in the Included Graph List described in §VI.B.

Syntax Rules:

- The Icon Family Name and Prototype Name are Variable Names.
- The Icon Family Tree is a Family Tree Table, and the entries shall conform to the syntax rules of Family Trees (see §V.A).
- In the table of Actual Type Arguments,
 - There shall be one row for each Formal Type Argument in the Icon's Prototype Form.
 - In the left column, the GUI shall list the Formal Type Arguments in the Prototype Form of the Icon.
 - In the right column, the operator shall enter a Type Name in each row.
- In the table of Actual Mode Arguments,
 - There shall be one row for each Formal Mode Argument in the Prototype Form.
 - In the Formal Height and Formal Base Types columns, the GUI shall list the respective Formal Height and Formal Base Type in the Formal Mode Arguments of the Icon's Prototype Form.
 - For each Actual Height column, the operator shall enter a non-negative integer.
 - For each Actual Base Type, the operator shall enter a Type Name.
- In the table of Actual GIP Bindings,
 - There shall be one row for each Formal GIP in the Prototype Form.
 - In the Formal GIPs column, the GUI shall list the respective Formal GIPs in the Icon's Prototype Form.
 - In each Fixed Family Tree, the syntax rules for a Family Tree apply (see §V.A).
 - The Value Field shall be either an expression or a NestedString (see the GSF Spec).
- All tables and fields below the heading "Initial Value" are used for initializing a Place.
 - If the Entity Type in the Icon's Prototype Form is "transition" or "inclgraph", then the Initial Value and all below it in this form shall be disabled and displayed in gray.
 - If the Entity Type is "gvar", then the Queue Token Index, together with its Lower and Upper Bounds shall be disabled and displayed in gray. Note: The Queue Token Index is the used to identify which token in the Queue is being initialized. For a Graph Variable, there is exactly one token, and so the token index is not appropriate.
 - The Lower Bound of the Queue Token Index (if applicable) shall be "1", read-only, as shown.
 - The Queue Token Index (if applicable) and the Fixed Table below it comprise a Family Tree.
 - The Queue Token Index and Fixed Table below it shall follow the syntax rules of a Family Tree.
 - Below the Queue Token Index, the number of rows in the Family Tree table shall equal the Actual Height specified in the Actual Mode Argument. (Recall that the Prototype Form for a place has exactly one Formal Mode Argument. Thus there is exactly one Actual Height, which specifies the height of the initial token(s) in the place.)
 - The Value Field shall be either an expression or a NestedString (see the GSF Spec).

Semantics Rules:

- The Icon Family Name shall be unique among Icon Family Names and Type Names in the GUI Graph. The Icon Family Name shall not conflict with any Prototype Name or Formal Name in the

GUI Graph Prototype Form. The GUI shall supply an initial unique name, which the operator may subsequently change.

- The Prototype Name shall be a Prototype Name that can be accessed as follows:
 - If the Call Form is associated with a Place Icon, the Prototype Name shall be the name of a Prototype Form accessible via the Prototypes Menu (see §II.B) whose Entity Type is either "queue" or "gvar".
 - If the Call Form is associated with a Transition Icon, the Prototype Name shall be the name of a Prototype Form accessible via the Prototypes Menu (see §II.B) whose Entity Type is "transition".
 - If the Call Form is associated with an Included Graph Icon, the Prototype Name shall be the name of a Prototype in the Included Graph List (see §VI.B).
- (Not enforceable by the GUI) No GUI Graph may contain itself as an Included Graph, either directly or indirectly.
- The Icon Family Tree shall follow the rules for a Family Tree (see §V.A). Each operand in the Lower and Upper Bound Expressions shall be a literal number, a Formal GIP of the Graph (i.e., in the GUI Graph Prototype Form), or an index in a row of the Icon Family Tree above the expression.
- Each Base Type in the Actual Type Arguments shall be either a language-defined type or a user-defined type that is defined in one of the files in the Type List (see §VI.D).
- In the table of Actual Mode Arguments, each Actual Base Type shall be either a language-defined type or a user-defined type that is defined in one of the files in the Type List (see §VI.D).
- In the table of Actual GIP Bindings, each operand in the Lower and Upper Bound expressions shall be a non-negative literal number, a Formal GIP in the GUI Graph, an index in the Icon Family Tree, or an index in a row above the expression in the same Family Tree.
 - If the operator enters a NestedString (see the GSF Spec) in the Value column, then the respective Family Tree shall be ignored. Each operand in the expressions in the NestedString shall be a literal, a Formal GIP in the GUI Graph Prototype Form, or an index in the Icon Family Tree. If an operand is a literal or a Formal GIP, then it shall be of the Base Type specified for the respective Formal GIP in the Icon's Prototype Form.
 - The operator may enter a Value that is the name of a Formal GIP of the GUI Graph, provided its height and base type respectively match the height and base type of the Formal GIP of the Icon's Prototype. In this case the respective Family Tree shall be ignored. (Note: This is not supported in the current version of the Translator Spec. ... RSS)
 - Except as described in the two immediately preceding bullets, the operator shall enter an expression in the Value column. Then each operand in the expression shall be a literal number, a Formal GIP in the GUI Graph Prototype Form, an index in the Icon Family Tree, or an index in the respective Family Tree.
- In the Initial Value,
 - If the Entity Type in the Prototype Form is "gvar", then the operator shall supply an initial value (Recall that in this case, the Queue Token Index is disabled).
 - If the Entity Type is "queue", then the operator has the option of supplying an initial value. The Queue Token Index identifies each of possibly many tokens. The Upper Bound specifies how many tokens, and shall be 0, by default, to indicate that the Queue is initially empty. In this case, all other fields under the heading "Initial Value" shall be left blank. The following semantic rules apply for a queue with at least one initial token and for a graph variable, which must be initialized with exactly one token.

- Each operand in the Lower and Upper Bound expressions shall be a literal number, a Formal GIP in the GUI Graph, an index in the Icon Family Tree, or an index in a row above the expression.
- If the operator enters a NestedString in the Value field, then the Family Tree, including the Queue Token index (if applicable), shall be ignored.
 - Each operand in the expressions in the NestedString shall be a literal number, a Formal GIP in the GUI Graph Prototype Form, or an index in the Icon Family Tree. If an operand is a literal or a Formal GIP, then it shall be of the Base Type specified for the place in the place's prototype form.
 - If the Entity Type is "queue", then the level of nesting in the NestedString shall equal one more than the Actual Token Height in the Actual Mode Argument.
 - If the Entity Type is "gvar", then the level of nesting in the NestedString shall equal the Actual Token Height in the Actual Mode Argument.
- Except as noted in the immediately preceding bullet and three sub-bullets, the operator shall enter an expression in the Value field. Then each operand in this expression shall be a literal number, a Formal GIP in the Graph Prototype Form, an index in the Icon Family Tree, or an index in the Initial Value Family Tree.

Mapping to EBNF:

- There is exactly one Call Form for each icon in the GUI Graph.
- The Information in the Call Form maps to the non-terminal Instance.
 - The Icon Family Name maps to the terminal *label*.
 - The Prototype Name maps to the Name in ProtCall.
 - The table titled Icon Family Tree is a Family Tree that maps to the list of Ranges.
 - In the Actual Type Arguments, each Base Type (second column) maps to the respective BaseType in ProtCall.
 - In the Actual Mode Arguments, each Actual Height and Actual Base Type map respectively to the TokHt and BaseType of the respective Mode in the FamInit.
 - In the Actual GIP Bindings,
 - Each Formal GIP maps to the respective Name in Bindings.
 - Note that the EBNF admits a choice of two forms for the Value in the Bindings. Thus there are two possible mappings to the Value in the Bindings, depending on the syntax of the Value.
 - If the Value field is a Nested String, then that Nested String is mapped to the Value.
 - If the Value field is an expression, then the Family Tree maps to the sequence of Ranges, and the expression in the Value field maps to the leaf Expr.
 - The entire Initial Value maps to the Value following the keyword "initval". The Queue Token Index and its Lower and Upper Bounds, together with the fixed table below it, comprise a Family Tree.
 - If the Value field is a Nested String, then that Nested String maps to the Value following the keyword "initval".
 - If the Value field is an expression, then the Family Tree maps to the sequence of Ranges in the Value, and the expression in the Value field maps to the Expr following the keyword "leaf".

VI.H Arc Form

The arc form is used to specify connections between families of ports of two icons. Note that each instance in an Icon Family with height > 0 may have a family of Ports with height > 0. For each family of Ports in an Icon Family, the family indices of both the Icon Family and the Port Family are listed, and the operator shall provide an integer expression which, when evaluated, assigns the value of the respective index. Each operand in the expressions shall be a literal number, a Formal GIP in the Prototype of the GUI Graph, or an index in an operator-defined nested loop. Each pass through the innermost loop establishes a single connection between two Ports that are identified by the respective values of the family indices. The operator may specify a predicate expression that tells whether the connection should be made.

Arc Form			
		<u>Nested Loop</u>	
	Index	Lower Bound	Upper Bound
□			

				<u>Connect</u>	
		Family Name	Index	Assigned Value	
From Icon:					
Output Port of the From Icon:					
To Icon:					
Input Port of the To Icon:					

When:

Before the Arc Form of an Arc can be opened, the Port Family Names of both icons must be identified. Note that the icons are already identified as the two icons at each end of the Arc on the screen. The Icon at the tail end of the Arc is the From Icon, and the Icon at the head end of the Arc is the To Icon. If the Port Family Names have not been identified,

- The GUI shall query the operator for the Port Names. Ideally, the GUI should supply the eligible port names, e.g., in a pop-up menu, and allow the operator to select the Port Family Name from the menu.
- As stated in the Connect table, the Port Family Name of the From Icon shall be an Output Port, and the Port Family Name of the To Icon shall be an Input Port.
- Additional requirements for valid selection of Ports:
 - The Mode (i.e., the Token Ht and Base Type) of the two Ports (specified in the respective Prototype Forms of the Icons - §VI.F) shall be the same.

- The Categories of the two Ports (specified in the respective Prototype Forms of the Icons - §VI.F) shall be different (i.e., one Port shall have Category "place", and the other shall be "transition").

This presupposes that the respective Icon's Call Form and Prototype Form have been created and filled out. The reason for this requirement is that the information in these forms is needed to display the correct table sizes and family indices for each Port Family.

Syntax Rules:

- The table following the heading Nested Loop is a Family Tree table, which shall follow the syntax rules of a Family Tree.
- In the Connect table,
 - For each Family Name, the Index and Assigned Value shall comprise a fixed table. The number of rows shall equal the respective family height. Specifically,
 - For each icon, the family height shall be the height of the Icon Family Tree in the Icon's Call Form. In the Index column of the Connect Table, the GUI shall automatically enter the indices of the Icon Family Tree.
 - For each Port, the family height shall be the height of the respective Port Family in the Icon's Prototype Form. In the Index column of the Connect Table, the GUI shall automatically enter the indices of the Port Family Tree.
 - In the Assigned Value column, for each Index, the operator shall enter an expression.
 - In the When field at the bottom of the Arc Form, the GUI shall provide the default value of "true". The operator may change it to an expression.

Semantics Rules:

- In the Nested Loop table, each expression for a Lower or Upper Bound shall be a literal number, a Formal GIP of the GUI Graph, or an index above the expression in the Nested Loop Table.
- In the expressions specifying Assigned Values, each operand shall be a literal number, a Formal GIP of the GUI Graph, or an index in the Nested Loop.
- An individual item in (an Icon or a Port) Family is identified by its Family Name together with the values of its indices.
 - No Port of any instance of an icon (i.e., node or included graph) may be connected more than once. The GUI shall enforce this requirement if both the icon family tree (specified in the call form) and the port family tree (specified in the prototype form) are empty (i.e., have height == 0). If either the icon or the port has a non-empty family tree, the GUI has insufficient information to enforce this requirement. The current implement of the GUI does not enforce this rule.
 - No Port of any Icon may be connected if its family is associated with a Graph Port.
- Each operand in the "When" expression shall be a literal number, a Formal GIP of the GUI Graph, or an index in the Nested Loop.
 - (Not enforceable by the GUI) The When expression shall return true or false for each pass in the innermost loop to indicate whether the specified connection should be made for the respective combination of assigned values of the indices in the Nested Loop.
- Each pair of connected ports shall have opposite category (as specified in the Prototype Form - §VI.F).
- Each pair of connected ports shall have the same Base Type (as specified in the Prototype Form).

- (Not enforceable by the GUI) Each pair of connected ports shall have the same Token Ht. NOTE: the operator may specify an expression for the Token Ht in the Prototype Form. The GUI has no way to evaluate this expression, and thus cannot make this check.

Mapping to EBNF:

- The information in the Arc Form shall be mapped to the non-terminal Arc.
 - The Family Tree information in the Nested Loop table shall be mapped to the sequence of Ranges following the keyword "arc".
 - For the From Icon and its Output Port Family (and likewise for the To Icon and its Input Port Family),
 - The From (To) Icon Family Name shall be mapped to the *label* of the first (second) non-terminal Connect.
 - The Assigned Values for each index of the From (To) Icon Family shall be mapped to the respective Expr in the sequence following the *label*.
 - The Output (Input) Port Family Name shall be mapped to the Name following the dot in the Connect.
 - The Assigned Values for each index of the Output (Input) Port Family shall be mapped to the respective Expr in the sequence following the Name.
 - The expression in the When field shall be mapped to the Expr following the keyword "when".

VI.I Icon Arc Form

The Icon Arc Form is opened via the Icon's pop-up Menu. The Icon Arc Form is read-only and gives Icon and Port Family Names of each arc connected to the selected Icon.

Icon Arc Form

Icon Family Name:

Input Ports

Input Port Family Name	From Icon Family Name	From Port Family Name	Open Arc Form
			<input type="checkbox"/>
			<input type="checkbox"/>

Output Ports

Output Port Family Name	To Icon Family Name	To Port Family Name	Open Arc Form
			<input type="checkbox"/>
			<input type="checkbox"/>

Syntax Rules:

- The Icon Family Name shall be displayed in the field at the top, as shown.

- For each arc whose head (tail) is connected to the Icon the GUI there shall be a row in the Input (Output) Ports table. Each row of this form shall display information in the respective Arc Form:
 - The Icon's Port Family Name shall be displayed in the left-most column.
 - The Icon Family Name of the other Icon (i.e., at the other end of the arc) shall be displayed in the second column.
 - The Port Family Name of the other Icon shall be displayed in the third column.
 - To view the full Arc Form read-only, the operator may click the respective button in the fourth column.

Appendix A

A.1 Pack Prototype Form

Prototype				
Prototype Name: <input style="width: 150px;" type="text" value="Pack"/>		Entity Type: <input style="width: 150px;" type="text" value="transition"/>		
<hr/> <u>Formal Type Arguments (Formal Name)</u> <hr/>				
<u>Formal Mode Arguments</u>				
Token Height (Formal Name)		Base Type (Formal Name)		
<input style="width: 150px;" type="text" value="height"/>		<input style="width: 150px;" type="text" value="base_type"/>		
<hr/> <u>Formal Graph Instantiation Parameters (GIPs)</u> <hr/>				
Name		Height (Number)		Base Type (Actual)
<hr/> <u>Input Ports</u> <hr/>				
Family Name	Category	Token Ht	Base Type	Family Tree
INPUT	transition	height	base_type	
READAMOUNT	transition	0	unsigned int	
READOFFSET	transition	0	unsigned int	
CONSUMEAMOUNT	transition	0	unsigned int	
CONSUMEOFFSET	transition	0	unsigned int	
<hr/> <u>Output Ports</u> <hr/>				
Family Name	Category	Token Ht	Base Type	Family Tree
OUTPUT	transition	height + 1	base_type	

The Prototype Form for the Pack Transition is read-only. Thus we show no add or delete buttons in any tables. The Formal Mode argument shows a "height" and "base_type". These are used in two ports: INPUT and OUTPUT. The Family Tree of every Port of the Pack is empty. Thus the cells in the Family Tree columns are blank. The button to open the Body is missing, because the Pack is a special transition and has a non-standard Transition Statement.

A.2 Unpack Prototype Form

Prototype				
Prototype Name: <input style="width: 150px;" type="text" value="Unpack"/>		Entity Type: <input style="width: 150px;" type="text" value="transition"/>		
<hr style="border: none; border-top: 1px solid black; margin: 5px 0;"/>				
Formal Type Arguments (Formal Name)				
<hr style="border: none; border-top: 1px solid black; margin: 5px 0;"/>				
<u>Formal Mode Arguments</u>				
Token Height (Formal Name)		Base Type (Formal Name)		
<input style="width: 150px;" type="text" value="height"/>		<input style="width: 150px;" type="text" value="base_type"/>		
<hr style="border: none; border-top: 1px solid black; margin: 5px 0;"/>				
<u>Formal Graph Instantiation Parameters (GIPs)</u>				
Name		Height (Number)	Base Type (Actual)	
<hr style="border: none; border-top: 1px solid black; margin: 5px 0;"/>				
<u>Input Ports</u>				
Family Name	Category	Token Ht	Base Type	Family Tree
<input style="width: 150px;" type="text" value="INPUT"/>	<input style="width: 100px;" type="text" value="transition"/>	<input style="width: 100px;" type="text" value="height + 1"/>	<input style="width: 100px;" type="text" value="base_type"/>	<input style="width: 100px;" type="text"/>
<hr style="border: none; border-top: 1px solid black; margin: 5px 0;"/>				
<u>Output Ports</u>				
Family Name	Category	Token Ht	Base Type	Family Tree
<input style="width: 150px;" type="text" value="OUTPUT"/>	<input style="width: 100px;" type="text" value="transition"/>	<input style="width: 100px;" type="text" value="height"/>	<input style="width: 100px;" type="text" value="base_type"/>	<input style="width: 100px;" type="text"/>
<input style="width: 150px;" type="text" value="PRODUCE"/>	<input style="width: 100px;" type="text" value="transition"/>	<input style="width: 100px;" type="text" value="0"/>	<input style="width: 100px;" type="text" value="unsigned int"/>	<input style="width: 100px;" type="text"/>

The Prototype Form for the Unpack Transition is read-only. Thus we show no add or delete buttons in any tables. The Formal Mode argument shows a "height" and "base_type". These are used in two ports: INPUT and OUTPUT. The Family Tree of every Port of the Unpack is empty. Thus the cells in the Family Tree columns are blank. The button to open the Body is missing, because the Unpack is a special transition and has a non-standard Transition Statement.

A.3 Standard Place Prototype Forms

Following are the Place Prototype Form for a standard Queue and a standard Graph Variable. Note that the Family Tree buttons are missing, because the Family Trees are empty to indicate that the Port Family Height is 0 and that the Ports are single Ports.

Place Prototype

Prototype Name: Entity Type:

Formal Mode Argument

Token Height (Formal Name)	Base Type (Formal Name)
height	base_type

Input Port

Family Name	Category	Token Ht	Base Type	Family Tree
INPUT	place	height	base_type	

Output Port

Family Name	Category	Token Ht	Base Type	Family Tree
OUTPUT	place	height	base_type	

Place Prototype

Prototype Name: Entity Type:

Formal Mode Argument

Token Height (Formal Name)	Base Type (Formal Name)
height	base_type

Input Port

Family Name	Category	Token Ht	Base Type	Family Tree
INPUT	place	height	base_type	

Output Port

Family Name	Category	Token Ht	Base Type	Family Tree
OUTPUT	place	height	base_type	

Appendix B

This Appendix gives an example of a GSF and the Forms for a graph.

B.1 Example GSF

```
capsule
{
  banner
  {
    filename {: /home/stevens/pgmt/gsf-draft/example1.gsf :}
    author   {: Dick Stevens :}
    revision {: version 1 :}
    purpose  {: to create an example for validating the translator :}
    graphtype 1;
  }

  exterior
  {
    prototype
      example1;
    gips
    {
      length : < 0, unsigned int >;
      width  : < 0, unsigned int >;
      breadth : < 0, unsigned int >;
    }
    inport
    {
      example1Input < 0, char >
        [0 : i : length - 1]
        [0 : j : 2*width - 1]
        [0 : k : breadth -1]
      category place;
    }
    outport
    {
      example1Output < 2, char >
      category place;
    }
  }
}
```

```

    }
input
{
    example1Input <=> Qin.INPUT;
}
output
{
    example1Output <=> Qout.OUTPUT;
}
}
transition
{
    prototype
    PassOn <T>;
    gips
    {
        how_long : < 0, unsigned int >;
        how_wide : < 0, unsigned int >;
    }
    inport
    {
        passIn < 0, T >
            [0 : r : how_long - 1]
            [0 : s : how_wide - 1] category transition;
        feedbackIn < 0, unsigned int > category transition;
    }
    outport
    {
        passOut < 2, T > category transition;
        feedbackOut < 0, unsigned int > category transition;
    }
    trstmt
    {:
        /* pass on the input token unchanged */
        /* set up the descriptor in the output workspace */
        passOut.assignFamily (passIn.getDescriptor());

        /* get the addresses of the leaf arrays */
        T * inputData = passIn.getData();
        T * outputData = passOut.getData();

        /* copy the data */

```

```

    for (int i = 0; i < passIn.getNumLeaves(); i++)
        outputData [i] = inputData [i];

    /* bump the counter */
    feedBackOut = feedBackIn + 1;
:}
}
queue
{
    prototype funnyQueue1 fmly (< height, base_type >);
    gips
    {
        how_broad : < 0, unsigned int >;
    }
    inport
    {
        INPUT < height, base_type >
        [0 : i : how_broad] category place;
    }
    outport
    {
        OUTPUT < height, base_type > category place;
    }
}
Qin
[0 : i : length - 1]
[0 : j : 2*width - 1]
[0 : k : breadth -1]
location < 100, 200 > =
place Queue fmly (< 0, char >);
QfeedBack
[1 : k : breadth]
location < 200, 100 > =
place Queue fmly (< 0, unsigned int >)
{
    initval
    {
        [1 : m : 1] leaf [ k ]
    }
};
Qout
location < 300, 200 > =

```

```

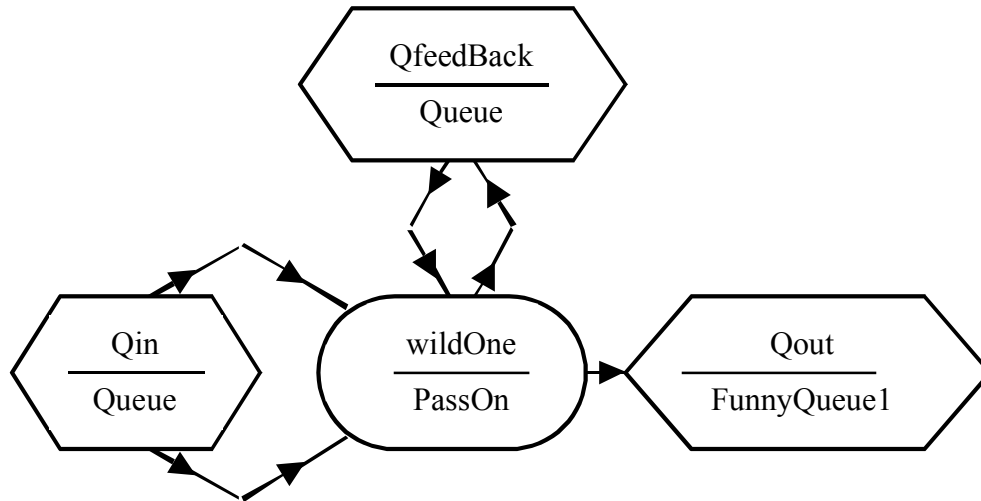
place funnyQueue1 fmly (< 2, char >)
{
  gips
  {
    how_broad = leaf [ breadth ];
  }
};
wildOne
[1 : k : breadth]
location < 200, 200 > =
transition PassOn <char>
{
  gips
  {
    how_wide = leaf [2*width];
    how_long = leaf [length];
  }
};
arc
[0 : i : length - 1]
[0 : j : 2*width - 1]
[0 : k : breadth - 1]
bends < 150, 175 >
{
/* Qin[i][j][k].OUTPUT -> wildOne[k+1].passIn[i][j] when (j/2)*2 == j
   */
  Qin[i][j][k].OUTPUT -> wildOne[k+1].passIn[i][j] when (j/2)*2 >= j
  /* i.e., when j is even */
}
arc
[0 : i : length - 1]
[0 : j : 2*width - 1]
[0 : k : breadth - 1]
bends < 150, 225 >
{
  Qin[i][j][k].OUTPUT -> wildOne[k+1].passIn[length - 1 - i][j]
  when (j/2)*2 < j /* i.e., when j is odd */
}
arc
[1 : k : breadth]
bends < 225, 150 >
{

```

```
        wildOne[k].feedBackOut -> QfeedBack[k].INPUT
    }
arc
    [1 : k : breadth]
    bends < 175, 150 >
    {
        QfeedBack[k].OUTPUT -> wildOne[k].feedBackIn
    }
arc
    [1 : k : breadth]
    {
        wildOne[k].passOut -> Qout.INPUT[k - 1]
    }
}
```

B.2 Graph Layout

This is the layout of the GSF in §B.1.



This example was designed more to exercise the GUI and the Translator than to do model a real application.

Every icon represents a family with height ≥ 0 . Qin is a family of queues with height = 3. QfeedBack and wildOne each have family height = 1. Qout has family height 0 and is thus a single queue. Qout is a queue with a family of input ports with height = 2.

PassOn defines the prototype for the transition wildOne. For each transition in the family called wildOne, there is a family of input ports called passIn with height = 0. These ports are connected in a strange way to the output ports of the queues in the family Qin. When the second index j is even, the ports are connected in the order specified by the other indices i and k . When the second index j is odd, the ports are connected so that the order is reversed for the first index i .

The tokens in the queues Qin and Qout have base type char. The tokens in the queues QfeedBack have base type unsigned int. Each transition in the family wildOne is connected to a queue in QfeedBack, which serves to count the executions of the transition. Each queue in QfeedBack is initialized with a token whose value is the index of the queue in the family.

In each execution of wildOne, one token is read from each input port connected to a queue in the family Qin. Those tokens are then assembled into a single token that has family height 2 and produced to the output port passOut, which is connected to one of the input ports of Qout. When a transition fires, it reads an unsigned integer from its member of the family of queues in QfeedBack, adds one, and produces the new value to the same queue.

Qout is a single queue with a family of input ports. Thus, Qout stores the tokens in the order that the wildOne transitions execute.

B.3 Example Forms

Following are the forms for the GSF in §B.1 above.

Graph Banner	
File Name:	<input type="text" value="/home/stevens/pgmt/gsf-draft/example1.h"/>
Author:	<input type="text" value="Dick Stevens"/>
Version:	<input type="text" value="version 1"/>
Purpose:	<input type="text" value="to create an example for validating the translator"/>
Main Graph:	<input checked="" type="radio"/>
Included Graph:	<input type="radio"/>

Included Graph List	
<input type="checkbox"/>	<u> Included Graph Prototype Name Full Path of Graph State File </u>

Type List	
<input type="checkbox"/>	<u> Full Path of Header File Defining User-Defined Class Definitions </u>

Graph Port Associations

<u>Graph Input Ports</u>		
Graph Input Port Family Name	Icon Family Name	Icon Input Port Family Name
example1Input	Qin	INPUT

<u>Graph Output Ports</u>		
Graph Output Port Family Name	Icon Family Name	Icon Output Port Family Name
example1Output	Qout	OUTPUT

Prototype Form

Prototype Name: Entity Type:

_____ Formal Type Arguments (Formal Name)

_____ Formal Mode Arguments
 Token Height (Formal Name) Base Type (Formal Name)

_____ Formal Graph Instantiation Parameters (GIPs)

Name	Height (Number)	Base Type (Actual)
length	0	unsigned int
width	0	unsigned int
breadth	0	unsigned int

_____ Input Ports

Family Name	Category	Token Ht	Base Type	Family Tree
example1Input	place	0	char	<input type="checkbox"/> <i>Exp Fam Tree</i>

_____ Output Ports

Family Name	Category	Token Ht	Base Type	Family Tree
example1Output	place	2	char	<input type="checkbox"/> <i>Exp Fam Tree</i>

Open Body

Family Tree

Family Name:

☐

	<u>Family Tree</u>	
Index	Lower Bound	Upper Bound
i	0	length - 1
j	0	2*width - 1
k	0	breadth - 1

Family Tree

Family Name:

☐

	<u>Family Tree</u>	
Index	Lower Bound	Upper Bound

Prototype Form

Prototype Name: Entity Type:

Formal Type Arguments (Formal Name)

Formal Mode Arguments
 Token Height (Formal Name) Base Type (Formal Name)

Formal Graph Instantiation Parameters (GIPs)

Name	Height (Number)	Base Type (Actual)
how long	0	unsigned int
how wide	0	unsigned int

Input Ports

Family Name	Category	Token Ht	Base Type	Family Tree
passIn	transition	0	T	<input type="checkbox"/> <i>Exp Fam Tree</i>
feedBackIn	transition	0	unsigned int	<input type="checkbox"/> <i>Exp Fam Tree</i>

Output Ports

Family Name	Category	Token Ht	Base Type	Family Tree
passOut	transition	2	T	<input type="checkbox"/> <i>Exp Fam Tree</i>
feedBackOut	transition	0	unsigned int	<input type="checkbox"/> <i>Exp Fam Tree</i>

Open Body

Family Tree

Family Name:

Family Tree

Index	Lower Bound	Upper Bound
r	0	how_long - 1
s	0	how_wide - 1

Family Tree

Family Name:

Family Tree
Index Lower Bound Upper Bound

Family Tree

Family Name:

Family Tree
Index Lower Bound Upper Bound

Family Tree

Family Name:

Family Tree
Index Lower Bound Upper Bound

Transition Statement

Transition Prototype Name:

```

/* pass on the input token unchanged */
/* set up the descriptor in the output workspace */
passOut.assignFamily (passIn.getDescriptor());

/* get the addresses of the leaf arrays */
T * inputData = passIn.getData();
T * outputData = passOut.getData();

/* copy the data */
for (int i = 0; i < passIn.getNumLeaves(); i++)
    outputData [i] = inputData [i];

/* bump the counter */
feedBackOut = feedBackIn + 1;
    
```

Place Prototype

Prototype Name: Entity Type:

Formal Mode Argument

Token Height (Formal Name)	Base Type (Formal Name)
height	base_type

Formal Graph Instantiation Parameters (GIPs)

Name	Height (Number)	Base Type (Actual)
<input type="checkbox"/> how_broad	0	unsigned int

Input Port

Family Name	Category	Token Ht	Base Type	Family Tree
INPUT	place	height	base_type	<input type="checkbox"/> <i>Exp Fam Tree</i>

Output Port

Family Name	Category	Token Ht	Base Type	Family Tree
OUTPUT	place	height	base_type	<input type="checkbox"/> <i>Exp Fam Tree</i>

Family Tree

Family Name:

	<u>Family Tree</u>	
	Lower Bound	Upper Bound
<input type="checkbox"/>	Index i	how_broad

Family Tree

Family Name:

	<u>Family Tree</u>	
	Lower Bound	Upper Bound
<input type="checkbox"/>	Index	

Call Form

Icon Family Name:
Prototype Name:

Icon Family Tree

<input type="checkbox"/>	Index	Lower Bound	Upper Bound
	i	0	length - 1
	j	0	2*width - 1
	k	0	breadth - 1

Actual Type Arguments

Formal Type	Base Type
-------------	-----------

Actual Mode Arguments

Formal Height	Actual Height	Formal Base Type	Actual Base Type
height	0	base_type	char

Actual GIP Bindings

Formal GIP	Family Tree	Value
------------	-------------	-------

Initial Value

Queue Token Index: Lower Bound: Upper Bound:

Index	Lower Bound	Upper Bound
-------	-------------	-------------

Value:

Call Form

Icon Family Name:
Prototype Name:

<u>Icon Family Tree</u>		
Index	Lower Bound	Upper Bound
k	1	breadth

<u>Actual Type Arguments</u>	
Formal Type	Base Type

<u>Actual Mode Arguments</u>			
Formal Height	Actual Height	Formal Base Type	Actual Base Type
height	0	base_type	unsigned int

<u>Actual GIP Bindings</u>		
Formal GIP	Family Tree	Value

Queue Token Index: Lower Bound: Upper Bound:

Index	Lower Bound	Upper Bound

Value:

Call Form

Icon Family Name:
Prototype Name:

Icon Family Tree
Index Lower Bound Upper Bound

Actual Type Arguments
Formal Type Base Type

Actual Mode Arguments

Formal Height	Actual Height	Formal Base Type	Actual Base Type
height	2	base_type	char

Actual GIP Bindings

Formal GIP	Family Tree	Value
how_broad	<i>Fix Fam Tree</i>	breadth

Initial Value
Queue Token Index: Lower Bound: Upper Bound:

Index	Lower Bound	Upper Bound

Value:

Call Form

Icon Family Name:
 Prototype Name:

	<u>Icon Family Tree</u>	
Index	Lower Bound	Upper Bound
k	1	breadth

<u>Actual Type Arguments</u>	
Formal Type	Base Type
T	char

<u>Actual Mode Arguments</u>			
Formal Height	Actual Height	Formal Base Type	Actual Base Type

<u>Actual GIP Bindings</u>		
Formal GIP	Family Tree	Value
how_wide	<i>Fix Fam Tree</i>	2*width
how_long	<i>Fix Fam Tree</i>	length

Initial Value

Queue Token Index: Lower Bound: Upper Bound:

Index	Lower Bound	Upper Bound
Value: <input type="text"/>		

Note: The Initial Value is gray to indicate it is disabled, because there is no initial value for a Transition Call Form.

Arc Form

Index	<u>Nested Loop</u> Lower Bound	Upper Bound
i	0	length - 1
j	0	2*width - 1
k	0	breadth - 1

From Icon:	Family Name	<u>Connect</u> Index	Assigned Value
	Qin	i	i
		j	j
		k	k
Output Port of the From Icon:	OUTPUT		
To Icon:	wildOne	k	k + 1
Input Port of the To Icon:	passIn	r	i
		s	j

When: $(j/2)*2 \geq j$ /* i.e., when j is even */

Arc Form

Index	<u>Nested Loop</u> Lower Bound	Upper Bound
i	0	length - 1
j	0	2*width - 1
k	0	breadth - 1

		<u>Connect</u>	
From Icon:	Family Name	Index	Assigned Value
	Qin	i	i
		j	j
		k	k
Output Port of the From Icon:	OUTPUT		
To Icon:	wildOne	k	k + 1
Input Port of the To Icon:	passIn	r	length - 1 - i
		s	j

When: $(j/2)*2 < j$ /* i.e., when j is odd */

Arc Form

<u>Nested Loop</u>		
Index	Lower Bound	Upper Bound
k	1	breadth

<u>Connect</u>			
	Family Name	Index	Assigned Value
From Icon:	wildOne	k	k
Output Port of the From Icon:	feedBackOut		
To Icon:	QfeedBack	k	k
Input Port of the To Icon:	INPUT		

When:

true

Arc Form

<u>Nested Loop</u>		
Index	Lower Bound	Upper Bound
k	1	breadth

<u>Connect</u>			
	Family Name	Index	Assigned Value
From Icon:	QfeedBack	k	k
Output Port of the From Icon:	OUTPUT		
To Icon:	wildOne	k	k
Input Port of the To Icon:	feedBackIn		

When:

true

Arc Form

	Index	<u>Nested Loop</u> Lower Bound	Upper Bound
□	k	1	breadth

	<u>Connect</u> Family Name	Index	Assigned Value
From Icon:	wildOne	k	k
Output Port of the From Icon:	passOut		
To Icon:	Qout		
Input Port of the To Icon:	INPUT	i	k - 1

When: true

B.4 C++ Output Code to be Generated by the Translator

Following is hand-written C++ code that should be generated by the Translator from the GSF in §B.1.

```
#ifndef example1_h
#define example1_h

/*****
File name: /home/stevens/pgmt/gsf-draft/example1.gsf
Author:    Dick Stevens
Revision:  version 1
Purpose:   to create an example for validating the translator
*****/

#include "GCL/GCL_Internal.h"
#include "primitives/external.h"

#ifndef PassOn_h
#define PassOn_h

template < class T >
class PassOn : public GCL_OrdTran
{
public:
    PassOn ( GCL_BaseType T_Datatype, unsigned int how_long, unsigned
int how_wide );
    ~PassOn ();
    void tranStmt ();
private:
    GCL_WorkSpace_T < T > _passIn;
    GCL_WorkSpace_T < unsigned int > _feedBackIn;
    GCL_WorkSpace_T < T > _passOut;
    GCL_WorkSpace_T < unsigned int > _feedBackOut;
};

template < class T >
PassOn <T> :: PassOn ( GCL_BaseType T_Datatype,
                    unsigned int how_long, unsigned int how_wide
)
: _passIn ( 0 + 2, T_Datatype),
  _feedBackIn ( 0 + 0, Machine::getMW_Datatype ( "unsigned int" ) ),
  _passOut ( 2 + 0, T_Datatype),
```

```

    _feedBackOut ( 0 + 0, Machine::getMW_Datatype ( "unsigned int" ) )
{
    GCL_Descriptor * descriptor;
    GCL_TranPort * tranport;
    {
        GCL_Descriptor * descriptor2;
        deque <GCL_Descriptor *> deque2;
        GCL_Descriptor * descriptor1;
        deque <GCL_Descriptor *> deque1;
        for (int r = 0; r <= how_long - 1; r++)
            {
                descriptor1 = new GCL_Descriptor ( 0, (how_wide - 1) - (0) +
1);
                deque2.push_back (descriptor1);
            }
        descriptor2 = new GCL_Descriptor ( deque2, 1, 0 );
        clearDeque ( deque2 );
        descriptor = descriptor2;
    }
    tranport = new GCL_TranInport_T < T >
        (descriptor, &_passIn, "passIn", 0, T_Datatype);
    _attachTranPort (tranport);

    descriptor = new GCL_Descriptor ();
    tranport = new GCL_TranInport_T < unsigned int >
        (descriptor, &_feedBackIn, "feedBackIn",
            0, Machine::getMW_Datatype ("unsigned int"));
    _attachTranPort (tranport);

    descriptor = new GCL_Descriptor ();
    tranport = new GCL_TranOutport_T < T >
        (descriptor, &_passOut, "passOut", 2, T_Datatype);
    _attachTranPort (tranport);

    descriptor = new GCL_Descriptor ();
    tranport = new GCL_TranOutport_T < unsigned int >
        (descriptor, &_feedBackOut, "feedBackOut",
            0, Machine::getMW_Datatype ("unsigned int"));
    _attachTranPort (tranport);
}

template < class T >

```

```

PassOn <T> :: ~PassOn () {}

template < class T >
void PassOn <T> :: tranStmt ()
{
    GCL_WorkSpace_T < T > & passIn = _passIn;
    unsigned int feedBackIn;
    _feedBackIn . getLeaf (feedBackIn);
    GCL_WorkSpace_T < T > & passOut = _passOut;
    unsigned int feedBackOut;

    /* pass on the input token unchanged */
    /* set up the descriptor in the output workspace */
    passOut.assignFamily (passIn.getDescriptor());

    /* get the addresses of the leaf arrays */
    T * inputData = passIn.getData();
    T * outputData = passOut.getData();

    /* copy the data */
    for (int i = 0; i < passIn.getNumLeaves(); i++)
        outputData [i] = inputData [i];

    /* bump the counter */
    feedBackOut = feedBackIn + 1;

    _feedBackOut . putLeaf (feedBackOut);
}

#endif PassOn_h

class example1 : public GCL_MainGraph
{
public:
    example1 ( unsigned int length,
               unsigned int width,
               unsigned int breadth,
               const CPRanks & cpRanks,
               const GraphRanks & graphRanks,
               const GraphPortAssignment & portAssignment,
               double maxLatency = 1.e18 );
    ~example1 ();
}

```

```

};

example1 :: example1 ( unsigned int length,
                      unsigned int width,
                      unsigned int breadth,
                      const CPRanks & cpRanks,
                      const GraphRanks & graphRanks,
                      const GraphPortAssignment & portAssignment,
                      double maxLatency = 1.e18 )
: GCL_MainGraph (cpRanks,
                 graphRanks,
                 portAsssignment,
                 "example1",
                 "version 1",
                 "/home/stevens/pgmt/gsf-draft/example1.gsf",
                 maxLatency )
{
  {
    GCL_Descriptor * descriptor3;
    deque <GCL_Descriptor *> deque3;
    GCL_Descriptor * descriptor2;
    deque <GCL_Descriptor *> deque2;
    GCL_Descriptor * descriptor1;
    deque <GCL_Descriptor *> deque1;
    for (int i = 0; i <= length - 1; i++)
      {
        for (int j = 0; j <= 2*width - 1; j++)
          {
            descriptor1 = new GCL_Descriptor
              (0, (breadth -1) - (0) + 1);
            deque2.push_back ( descriptor1 );
          }
        descriptor2 = new GCL_Descriptor ( deque2, 1, 0);
        clearDeque ( deque2 );
        deque3.push_back ( descriptor2 );
      }
    descriptor3 = new GCL_descriptor ( deque3, 2, 0);
    clearDeque ( deque3 );
    _descriptor = descriptor3;
  }
  _stringIDs = new GCL_StringIDs ( "example1Input",
                                   "GCL_GraphInport_T<char>",

```

```

        _graphClassName,
        _graphVersion,
        _graphFileName );
_graphobjhdl = new GCL_GraphObjHdl ( (GCL_Graph *) this,
        GRAPHPORT,
        _descriptor,
        _stringIDs );

{
    GCL_GraphInport_T < char > * example1Input;
    for (int j = 0; j < _descriptor -> getLeafCount(); j++)
    {
        example1Input = new GCL_GraphInport_T<char>
            (0, Machine::getMW_Datatype("char"));
        _graphobjhdl -> addGraphObj ( (GCL_GraphObj *) example1Input );
    }
}

_descriptor = new GCL_Descriptor ();
_stringIDs = new GCL_StringIDs ( "example1Output",
        "GCL_GraphOutputport_T<char>",
        _graphClassName,
        _graphVersion,
        _graphFileName );
_graphobjhdl = new GCL_GraphObjHdl ( (GCL_Graph *) this,
        GRAPHPORT,
        _descriptor,
        _stringIDs );

{
    GCL_GraphOutputport_T < char > * example1Output;
    for (int j = 0; j < _descriptor -> getLeafCount(); j++)
    {
        example1Output = new GCL_GraphOutputport_T<char>
            (2, Machine::getMW_Datatype("char"));
        _graphobjhdl -> addGraphObj ((GCL_GraphObj *) example1Output);
    }
}

{
    GCL_Descriptor * descriptor3;
    deque <GCL_Descriptor *> deque3;
    GCL_Descriptor * descriptor2;
    deque <GCL_Descriptor *> deque2;
}

```

```

GCL_Descriptor * descriptor1;
deque <GCL_Descriptor *> deque1;
for (int i = 0; i <= length - 1; i++)
{
    for (int j = 0; j <= 2*width - 1; j++)
    {
        descriptor1 = new GCL_Descriptor
            (0, (breadth -1) - (0) + 1);
        deque2.push_back ( descriptor1 );
    }
    descriptor2 = new GCL_Descriptor ( deque2, 1, 0);
    clearDeque ( deque2 );
    deque3.push_back ( descriptor2 );
}
descriptor3 = new GCL_descriptor ( deque3, 2, 0);
clearDeque ( deque3 );
_descriptor = descriptor3;
}
_stringIDs = new GCL_StringIDs ( "Qin",
                                "GCL_Queue_T<char>",
                                _graphClassName,
                                _graphVersion,
                                _graphFileName );
_graphObjHdl = new GCL_GraphObjHdl ( (GCL_Graph *) this,
                                     PLACE,
                                     _descriptor,
                                     _stringIDs );
{
    GCL_Queue_T < char > * Qin;
    for (int i = 0; i <= length - 1; i++)
    {
        for (int j = 0; j <= 2*width - 1; j++)
        {
            for (int k = 0; k <= breadth -1; k++)
            {
                Qin = new GCL_Queue_T < char >
                    ( 0, Machine::getMW_Datatype("char") );
                _graphObjHdl -> addGraphObj ( (GCL_GraphObj *) Qin);
            }
        }
    }
}
}

```

```

{
    GCL_Descriptor * descriptor1;
    deque <GCL_Descriptor *> deque1;
    descriptor1 = new GCL_Descriptor ( 1, (breadth) - (1) + 1 );
    _descriptor = descriptor1;
}

_stringIDs = new GCL_StringIDs ( "QfeedBack",
                                "GCL_Queue_T<unsigned int>",
                                _graphClassName,
                                _graphVersion,
                                _graphFileName );
_graphObjHdl = new GCL_GraphObjHdl ( (GCL_Graph *) this,
                                     PLACE,
                                     _descriptor,
                                     _stringIDs );

{
    GCL_Queue_T < unsigned int > * QfeedBack;
    for (int k = 1; k <= breadth; k++)
    {
        {
            GCL_Descriptor * descriptor1;
            deque <GCL_Descriptor *> deque1;
            descriptor1 = new GCL_Descriptor (1, (1) - (1) + 1);
            _descriptor = descriptor1;
        }

        GCL-Token_T < unsigned int > * genericToken
            = new GCL-Token_T < unsigned int > (_descriptor);
        unsigned int * tokenValue = genericToken -> getLeafArray();
        { unsigned int leafIndex = 0;
          for (int m = 1; m <= 1; m++)
          {
              tokenValue [leafIndex++] = k;
          }
        }
        QfeedBack = new GCL_Queue_T < unsigned int >
            (0, Machine::getMW_Datatype ("unsigned int"), genericToken);
        _graphObjHdl -> addGraphObj ( (GCL_GraphObj *) QfeedBack);
    }
}

```

```

_descriptor = new GCL_Descriptor ();
_stringIDs = new GCL_StringIDs ( "Qout",
                                "GCL_Queue_T<char>",
                                _graphClassName,
                                _graphVersion,
                                _graphFileName );
_graphObjHdl = new GCL_GraphObjHdl ( (GCL_Graph *) this,
                                     PLACE,
                                     _descriptor,
                                     _stringIDs );

{
    GCL_Queue_T < char > * Qout;
    unsigned int how_broad = breadth;
    GCL_Descriptor * inportDescriptor;
    GCL_Descriptor * outportDescriptor;

    {
        GCL_Descriptor * descriptor1;
        deque <GCL_Descriptor *> deque1;
        descriptor1 = new GCL_Descriptor (0, (how_broad) - (0) + 1);
        inportDescriptor = descriptor1;
    }
    outportDescriptor = new GCL_Descriptor ();
    Qout = new GCL_Queue_T < char >
        ( 2, Machine::getMW_Datatype("char") ),
        inportDescriptor, outportDescriptor );
    _graphObjHdl -> addGraphObj ( (GCL_GraphObj *) Qout);
}

{
    GCL_Descriptor * descriptor1;
    deque <GCL_Descriptor *> deque1;
    descriptor1 = new GCL_Descriptor (1, (breadth) - (1) + 1);
    _descriptor = descriptor1;
}
_stringIDs = new GCL_StringIDs ( "wildOne",
                                "PassOn<char>",
                                _graphClassName,
                                _graphVersion,
                                _graphFileName );

```

```

_graphObjHdl = new GCL_GraphObjHdl ( (GCL_Graph *) this,
                                     TRAN,
                                     _descriptor,
                                     _stringIDs );
{
    PassOn<char> * wildOne;
    unsigned int how_long;
    unsigned int how_wide;
    for (int k = 1; k <= breadth; k++)
        {
            how_wide = 2*width;
            how_long = length;
            wildOne = new PassOn<char> ( Machine::getMW_Datatype
("char"),
                                     how_long, how_wide );
            _graphObjHdl -> addGraphObj ( (GCL_GraphObj *) wildOne);
        }
}

_associateGraphInport ("example1Input", "Qin", "INPUT");
_associateGraphOutport ("Qout", "OUTPUT", "example1Output");

_portIndexResize (3, 0, 1, 2);
{
    for (int i = 0; i <= length - 1; i++)
        {
            for (int j = 0; j <= 2*width - 1; j++)
                {
                    for (int k = 0; k <= breadth - 1; k++)
                        {
                            _fromObjIndices [0] = i;
                            _fromObjIndices [1] = j;
                            _fromObjIndices [2] = k;
                            _toObjIndices [0] = k + 1;
                            _toPortIndices [0] = i;
                            _toPortIndices [1] = j;
                            if ( (j/2)*2 >= j ) /* i.e., if j is even */
                                _connectPorts ( "Qin", _fromObjIndices,
                                                "OUTPUT", _fromPortIndices,
                                                "wildOne", _toObjIndices,
                                                "passIn", _toPortIndices );
                        }
                }
        }
}

```

```

    }
  }
}

_portIndexResize (3, 0, 1, 2);
{
  for (int i = 0; i <= length - 1; i++)
  {
    for (int j = 0; j <= 2*width - 1; j++)
    {
      for (int k = 0; k <= breadth - 1; k++)
      {
        _fromObjIndices [0] = i;
        _fromObjIndices [1] = j;
        _fromObjIndices [2] = k;
        _toObjIndices [0] = k + 1;
        _toPortIndices [0] = length - 1 - i;
        _toPortIndices [1] = j;
        if ( (j/2)*2 < j ) /* i.e., if j is odd */
          _connectPorts ( "Qin", _fromObjIndices,
                        "OUTPUT", _fromPortIndices,
                        "wildOne", _toObjIndices,
                        "passIn", _toPortIndices );
      }
    }
  }
}

_portIndexResize (1, 0, 1, 0);
{
  for (int k = 1; k <= breadth; k++)
  {
    _fromObjIndices [0] = k;
    _toObjIndices [0] = k;
    _connectPorts ("wildOne", _fromObjIndices,
                  "feedBackOut", _fromPortIndices,
                  "QfeedBack", _toObjIndices,
                  "INPUT", _toPortIndices );
  }
}

```

```

_portIndexResize (1, 0, 1, 0);
{
    for (int k = 1; k <= breadth; k++)
        {
            _fromObjIndices [0] = k;
            _toObjIndices [0] = k;
            _connectPorts ("QfeedBack", _fromObjIndices,
                           "OUTPUT", _fromPortIndices,
                           "wildOne", _toObjIndices,
                           "feedBackIn", _toPortIndices );
        }
}

_portIndexResize (1, 0, 0, 1);
{
    for (int k = 1; k <= breadth; k++)
        {
            _fromObjIndices [0] = k;
            _toPortIndices [0] = k - 1;
            _connectPorts ("wildOne", _fromObjIndices,
                           "passOut", _fromPortIndices,
                           "Qout", _toObjIndices,
                           "INPUT", _toPortIndices );
        }
}

_initializeGraph();
}

example1 :: ~example1() {}

#endif example1_h

```