A Petri Net-based Environment for GUI Design

Xiaosong Li, Warwick B. Mugridge, and John G. Hosking

Department of Computer Science University of Auckland
Private Bag 92019, Auckland, New Zealand
{x_li,rick,john}@cs.auckland.ac.nz

ABSTRACT

We describe PUIST, a visual environment for graphical user interface specification and prototyping. PUIST uses a Petri net notation, with a declarative means of defining nets which have complex, yet regular interconnections. This significantly improves the understandability of large specifications, permitting PUIST to be used for complex interface component specification, analysis and prototyping.

1. INTRODUCTION

PUIST (Petri net based graphical User Interface Specification Tool) is a visual environment for specifying graphical user interfaces. Petri nets [1] are used as the specification notation, with the ability to specify both static and dynamic properties of the GUI. Petri net nodes may be associated with GUI objects, such as windows, and buttons. PUIST also supports implementation and execution of Petri net specifications, permitting prototype user interfaces to be tested concurrently with their specification. PUIST is aimed at rapid prototyping for early checking of functional requirements for GUI design.

The idea of associating Petri net nodes with GUI objects comes from the Trellis model of hypertext [2]. In this model, a hypertext document consists of a Petri net representing the document's linked structure, several sets of "human-consumable" components (contents, windows, and buttons), and two collections of mappings, termed projections, between the Petri net, the human-consumables, and the display mechanisms. A place may be mapped to a set of document content or a set of windows. A transition may be mapped to a set of buttons. PUIST adopts a similar approach of associating GUI objects to places and transitions.

PUIST, however, extends the power of Petri nets to increase the readability and understandability of Petri nets for large specifications. *Subnet types* are introduced for defining Petri nets which have complex yet regular pattern structures. The subnet type formalism provides a novel and useful form of parametric abstraction permitting straightforward reuse of specifications. Subnet instantiation generates a Petri net (subnet) from its type definition automatically.

The paper commences with a description of the basic PUIST notation, followed by a description of the subnet specification notation, illustrated by an example. This is followed by a description of analysis support, discussion and conclusions.

2. BASIC NOTATIONS

PUIST's Petri net notations are based on place- transition nets [1,3].

Definition 1: Formally, a Petri net is a 5-tuple,
PN = (P, T, F, W, M) where:
$P = \{p_1, p_2, ..., p_m\}$ is a finite set of places,
$T = \{t_1, t_2, ..., t_n\}$ is a finite set of transitions,
$F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs,
$W: F \rightarrow \{1, 2, 3, ...\}$ is a weight function,
$M: P \rightarrow \{1, 2, 3, ...\}$ is the token marking,
$P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

The state of a Petri net is changed according to the following transition *firing rules*:

- A *transition* t is said to be *enabled* if each input place p of t has at least w(p,t) tokens, where w(p,t) is the *weight* of the arc from p to t.
- An enabled transition will fire when its associated *event* occurs.
- *Firing* transition t removes w(p,t) tokens from each input place p of t, adding w(t,p') tokens to each output place p', w(t,p') being the weight of the arc from t to p'.

PUIST permits places and transitions to be associated with GUI component objects. GUI objects are classified into *action* and *base* objects. Action objects, such as buttons or menu items, are associated with transitions. GUI events, such as a key press or mouse click, fire the associated transition. Base objects, such as dialogue boxes or windows, are associated with places. When the place holds a token, the corresponding base object is displayed (window) or enabled (menu). Removal of all tokens closes or disables the base object. The Petri net execution semantics thus specify the GUI's dynamic behaviour. We extend Definition 1 to include these GUI objects:

0-7803-4053-1/97/$10.00 © 1997 IEEE

Definition 2.: The PUIST model is a 9- tuple, (P, T, F, W, M, A_O, B_O, S_a, S_b) where:

P, T, F, W and M are the same as in Definition 1.

A_O is a set of action objects.

B_O is a set of base objects.

S_a is a 1 - 1 mapping $T \rightarrow A_O$.

S_b is a 1 - 1 mapping $P \rightarrow B_O$.

S_a associates one transition with a single object in A_O. Several types of object can occur in A_O, including buttons, menu items, and simple_trans objects. The latter are virtual action object unrelated to GUI objects. S_b associates one place with a single object in B_O. Several types of object can occur in B_O, including windows, menus, and virtual (non GUI) objects [4].

Enabling a transition implies enabling the associated action object. For example, a menu item is enabled by enabling its associated transition. Firing a transition implies that the user has acted on the action object. Putting a token into a place implies showing the base object. Emptying a place implies hiding the base object.

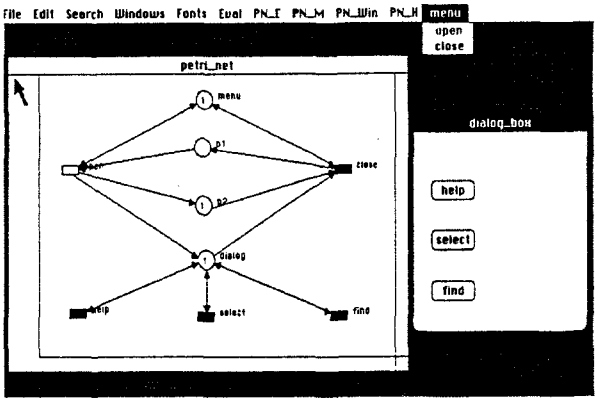

Fig.1: A Petri net specifying one menu and one dialog box

Fig. 1 shows a Petri net specifying one menu and one dialog box, The menu has two menu items, open and close, which control the dialog box. Transitions open and close are mapped to menu items open and close respectively. Transition open is disabled, thus menu item open is disabled. Menu item close is enabled because transition close is enabled. Buttons {help, select, find} have a similar correspondence with transitions {help, select, find}. The dialog_box and menu are associated with places dialog and menu respectively. The dialog_box is open as there is one token in place dialog. Likewise the menu is enabled. Places p1 and p2 are mapped to two virtual GUI objects respectively. They are used to ensure that only one of the menu items open and close is enabled at any one time. Selecting close empties the dialog, menu, and p2 places, and places a token in p1 and back in menu. Hence the dialog is closed, the menu remains enabled, the close item disabled, and the open item enabled (p1 has a token).

In addition to definitions 1 and 2 and the transition firing rules, PUIST has three specialised components. *Emptying arcs* (▪ ▪▸), are specialised place-to-transition arcs: an emptying arcs does not affect the enabling of the transition. However, when the transition fires, the place is emptied. *Inhibitor arcs* (~~●) also connect a place to a transition: only when the place is empty can the transition be enabled. As long as an *autofiring transition* is enabled it fires, without any event. Details of the PUIST notation may be found in [4, 5].

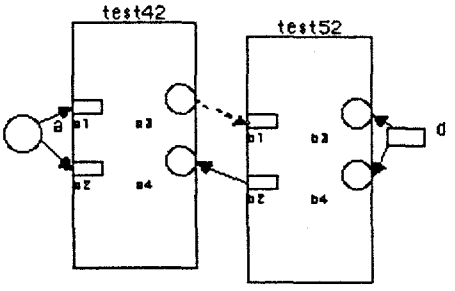

Fig.2: Example subnet icons

3. SUBNETS

A *subnet type* defines an abstract Petri net structure. Given instantiation parameters, it can generate a family of Petri net pieces, called *subnets*, based on the same abstract pattern. Visually, a subnet is represented as a rectangular icon with named connection points along its edges, representing internal places and transitions accessible from outside the subnet. Fig. 2. shows two subnet icons, connected together and to an additional place and transition.

3.1 Subnet type specifications

Fig. 3 shows a subnet type specification. This defines some of the behavioural semantics of a set of folders in a desktop window. In this example, folders may be selected (hence highlighted) by clicking on them. Selecting one folder causes any other selected folder to be deselected. In Fig. 3 the subnet type takes four arguments, representing the name of the window, the x-y location of the first folder, and a list of folder specifications. Subnet types may include a number of *cases*, differentiated by their parameter lists, specified in a box at the top. Prolog-style pattern matching is used to select between the cases when a specification is instantiated. The first case, exclusive_folders[1], matches when the folder list is empty. The second case matches the head and tail of a non-empty list.

The internal specification of each case can contain any PUIST element (places, transitions, and arcs) together with (possibly recursive) subnet instantiations, *instantiation bindings* and *connection specifiers*. Instantiation bindings (▭) use formal parameters of a subnet type to specify embedded transitions, places, and subnet type instantiations. For example, the instantiation binding attached to the top transition binds the name of the folder to

be the head of the folder specification list (and also specifies other initialisation parameters for the GUI object). The binding for the recursive application supplies the window name and remaining folder specifiers as arguments.
Connection specifiers specify external connections to the subnet, i.e. which internal components are connectable from outside the subnet, and their relative positions on the edges of the subnet icon. Fig. 3 shows two connection specifiers: a list of transitions ( ) on the top edge, and a list of places ( ) on the bottom. The actual transitions and places appearing in an instantiation of the subnet are determined by the internal connection "wiring" specified by the thick grey lines. If there are no such "wires", such as in exclusive_folders[1], the lists are empty (ie no transitions or places are exported from the instantiation). The order of the connection points shown on the subnet icon follows the order of attachment of the internal "wiring" to the attachment points. Thus, in exclusive_folders[2] the exported transition list consists of the transition defined in the case, followed by the list of like transitions formed from the recursive application of exclusive_folders. Connection specifiers are identical for all cases in one subnet type specification.
The exclusive_folders[2] case consists of a transition, a place, a recursive instantiation of the folders subnet type and several arcs. The transition represents one folder, the action of which is generated when the folder is clicked on. The place represents the selection status of the folder. If it has a token, the folder is selected and thus highlighted. The recursive instantiation represents a collection of other folder controllers of the same form. The two emptying arcs ensure that if any folder is selected, ie its transition is fired, the selection status of all other folders is emptied, hence deselecting them. This is an example of a mutually exclusive Petri net pattern. Similar patterns occur in mutually exclusive menu selections, selection of currently active windows, etc.
Subnet types can be used in the definition of other subnet types, either recursively, as in exclusive_folders, or as conventional internal components, as in the desktop subnet type shown in Fig. 4. The latter consists of a place representing a window, and a collection of exclusive folders, contained in the window.

3.2 Subnet instantiation

Instantiation of a subnet is done by supplying actual parameter values to a Petri net type specification via a dialog box. Figure 5 shows a Petri net generated by instantiation of the desktop subnet type (and hence the internal exclusive_folders subnet) for a window, *desk*, with three folders, *folder1, folder2, folder3*. The instantiation parameters are {desk, [folder1, folder2, folder3]}. The corresponding desktop window is shown in Fig 6.

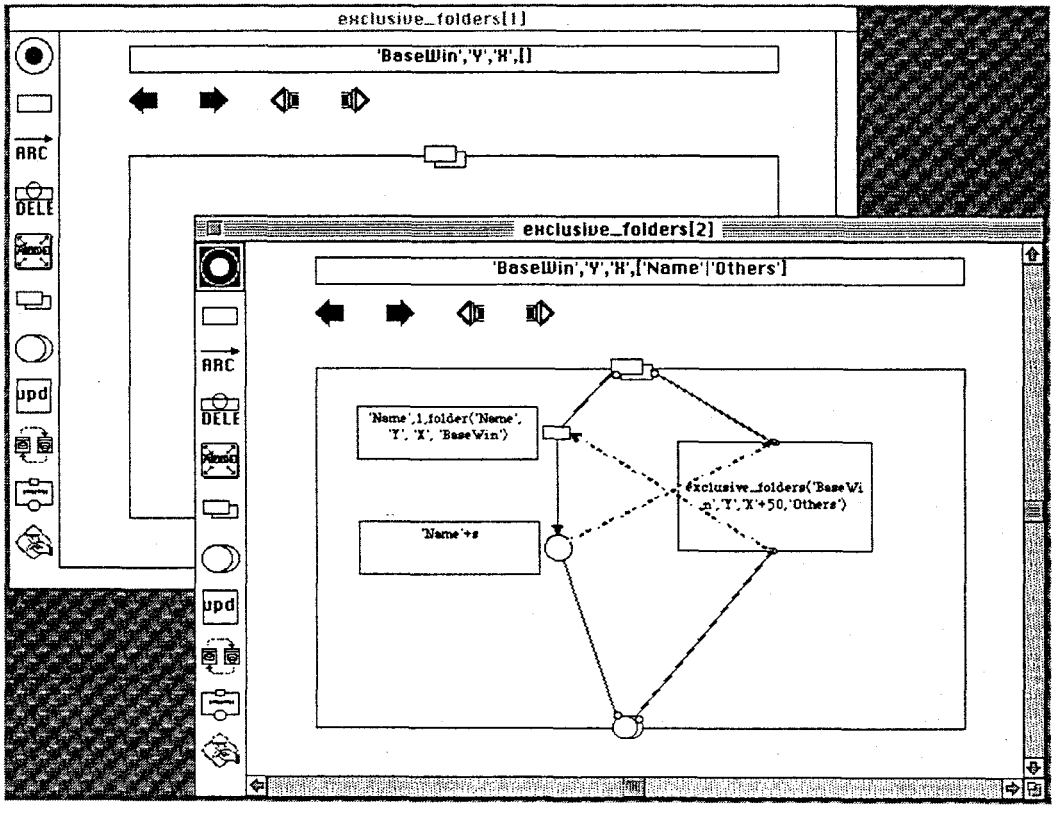

Fig. 3: Mutually exclusive folders subnet type specification

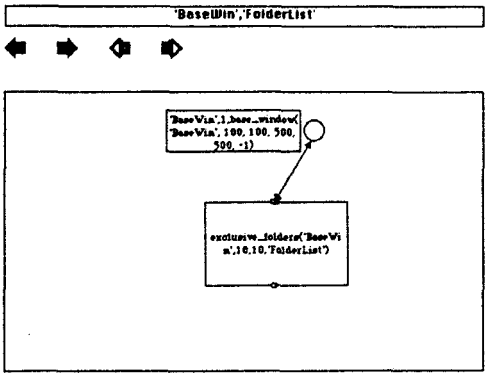

Fig. 4: Desktop subnet type

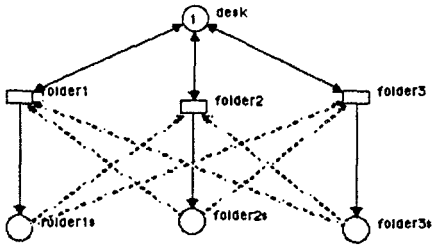

Fig. 5: Petri net generated from Fig. 4

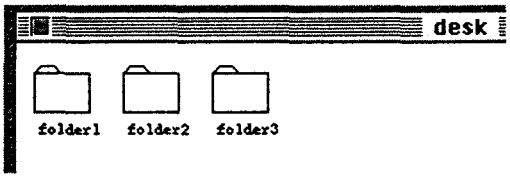

Fig 6: Desktop corresponding to Fig. 5

4. EXTENDED FOLDER SYSTEM

Consider the following extension to the folder example. Selected folders may be opened, by clicking on them a second time. This opens an associated window, which may itself contain folders, deselecting the folder in the parent window. All but the initial window may be closed, reselecting the parent folder.

A specification for the desktop system can be developed in a modular fashion using subnets. Fig. 7 shows subnet type folders_module, defining semantics for the folders in one window. Only the recursive case is shown; the base case is empty. This extends the mutual exclusion pattern of Fig. 3. An autofiring transition empties the selection status after a second click on the folder (due to the weight 2 arc). This passes a token to the place connection specifier of an instantiation of a window type (Fig. 8) causing the window associated with the folder to open. The transition connection of the window, which fires when the window is closed, is wired to the selection status of the folder, reselecting the folder when the window closes. The folder list parameter is extended. Each folder has a name, and collection of subfolders (ie the folders in its associated window).

The window specification is shown in Fig. 8. This consists of a place, representing the window, a transition, representing the close window action, and an instantiation of folder_module, specifying the folders in the window. The place-transition arc causes the window to close when the close action occurs, by emptying the token from the place. The two way arc from the place to the window connection specifier enables the child folders when the place holds a token.

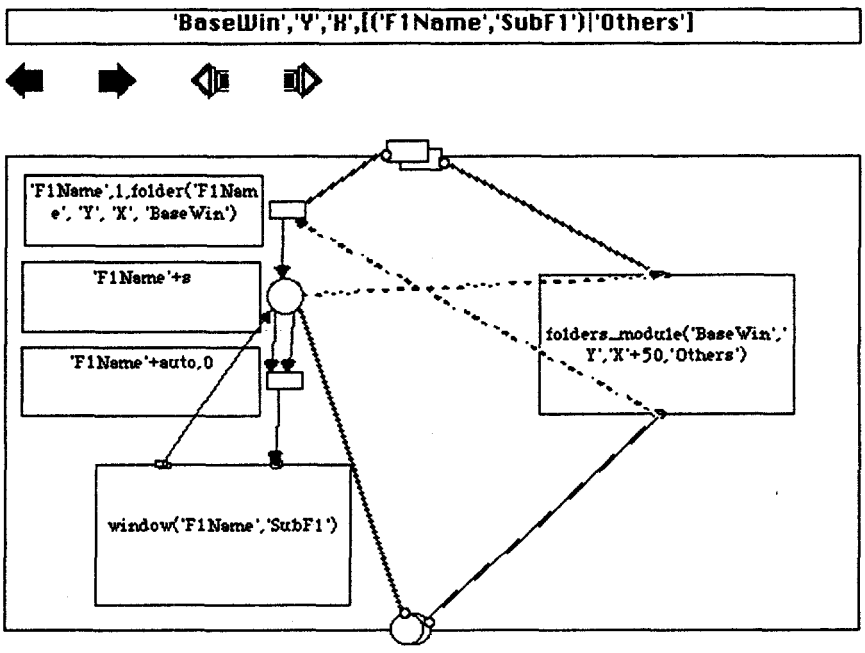

Fig. 7: Recursive case of folders_module subnet type

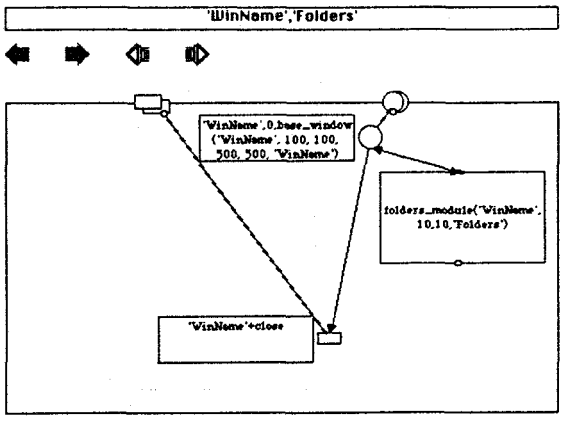

Fig. 8 window subnet type

The two subnet types are thus mutually recursive: each window has an associated, possibly empty, collection of folders. Each folder has a child window. The base case of the recursion is the empty folders_module case.

To complete the folder example specification, we must specify the initial window. This window is not allowed to close. The desktop subnet type specification in Fig. 9 shows how this is achieved, using a place with a token, a transition, and an instantiation of the window subnet type. The place and transition are wired to the window status place to ensure the window is open. An inhibitor arc from the place to the window close connection specifier ensures the window close action is disabled in this initial window. An alternative approach would be to use an explicit window place GUI object and window close transition, and include an instantiation of folders_module rather than window in the desktop subnet type specification.

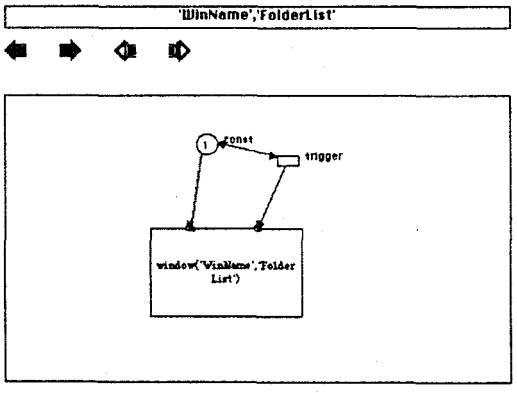

Fig. 9 Extended desktop subnet type

Fig. 10 shows an expanded instantiation of the desktop subnet specification with parameters d7, [(n1, [(n3, [])]), (n2, [])], generated automatically by PUIST. Note the complexity of Fig. 10, in comparison with the compact abstractions of Figs 7-9. Fig. 11 shows the generated desktop prototype. The prototype and Petri net are live, allowing the specification to be tested.

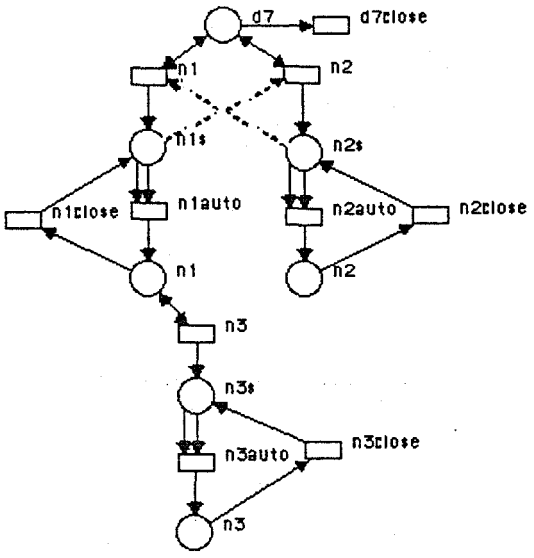

Fig. 10: Instantiation of desktop net

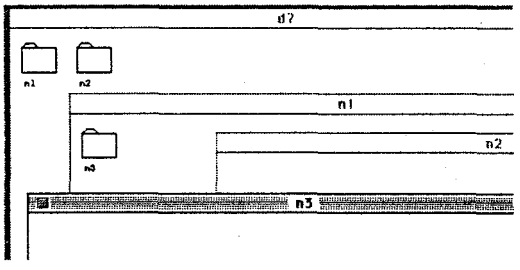

Fig. 11: instantiated desktop

5. ANALYSIS TOOLS

There are two main Petri net analysis methods: reachabilty trees and incidence matrices [1,3]. The use of self-loops (transitions with inputs and outputs from the same place) and the three special components (Section 2) make many PUIST Petri nets unsuitable for incidence matrix analysis, hence this method is not supported by PUIST.

PUIST does support reachability tree analysis. The reachability tree represents the reachability set of a Petri net [3]. Figure 12 shows the reachability tree for the Petri net in Fig. 1. produced by PUIST. This is based on the algorithm given in Section 4.2 of [3], with special consideration for subnets and the three special components. Subnet places and transitions need to be added to the places and the transitions in the original net as the input to the algorithm. Inhabitor arcs modify the test for an enabled transition. Empty arcs modify the way the marking state is updated after transition firing. The autofiring transition affects the algorithm more seriously. For a stable marking state, an autofiring transition only fires passively, accompanying the firing of another driving transition.

From the reachability tree of Fig. 12 we can obtain the following properties for the Petri net of Fig 1: conservation, boundedness, liveness, and controllability [1,3]. The net is conservative and bounded. The net is live,

which means it is possible to fire any transition of the net by progressing through some firing sequence. This is important to a GUI as we expect all buttons, menu_items, etc to be useful. The net is completely controllable, which means any marking is reachable from any other marking, this makes the system more user friendly because a user will not get lost from any state of the system. More useful ways off mapping Petri net properties to GUI semantics is the focus of current work.

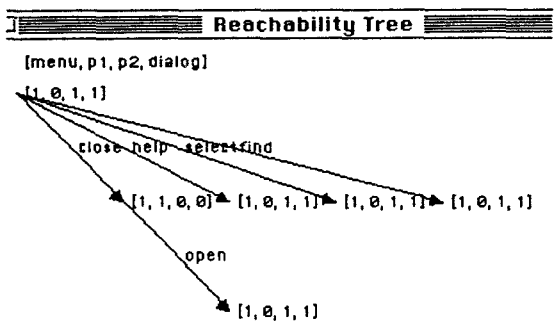

Fig 12. Reachability graph

6. DISCUSSION AND CONCLUSIONS

Several research groups are investigating the use of Petri net specifications of GUIs. Most notable is the work of Palanque and his associates developing the PNO (Petri Nets with Objects) and ICO (Interactive Cooperative Objects) systems [6,7]. These are based on high-level Petri nets, using typed tokens for communication. However, these systems lack both modularity and the ability to automatically generate low level Petri nets as is provided by PUIST. Other groups have been investigating software specifications using Petri nets [9,10]. These systems provide hierarchical structures and high level abstractions, and they support the execution of the Petri nets. However their abstractions lack the power of PUIST's subnet type and subnet generation.

We have presented and implemented a visual language for specifying GUIs using an extended Petri net notation. The basic notation allows simple GUI components to be rapidly specified and prototyped. The subnet formalism provides a novel and useful form of parametric abstraction permitting straightforward reuse of specifications. The visual formalism allows both hierarchical and recursive composition of GUI components to be specified, overcoming the problems of diagram complexity that typically arise from the use of larger Petri nets.

It is clear that subnet type specifications can be used to modularise Petri net specifications so that each component is of a managable size. When the subnet type formalism is used for a GUI design, either a top-down or bottom-up design strategy can be employed. In section 4 a bottom-up strategy is used. PUIST allows an empty subnet type to be created as a stub for the early stage of a top-down design and to be fully specified in the later stage of the design. However, as with any new type of abstraction, it is not immediately obvious how to best make use of subnet type specifications. For example, how "wide" should the interface to a subnet be before resulting interconnections reduce the effectiveness of the modularity. This will require experience to determine the best way to modularise patterns of structure and behaviour in Petri net form, just as it took some time before appropriate rules for using procedural abstraction or class abstraction were developed for imperative and object-oriented programming respectively.

We are currently constructing a library of standard subnet types, and extending the range of system specifications beyond that of simple GUI componentry. We are also investigating the application of formal analysis techniques, previously developed for Petri nets [1,3], to the analysis of GUIs specified using PUIST, in similar manner to [2]. This will extend the reachability graph work presented here. Other possibilities for the application of PUIST's subnet formalism are also being investigated.

REFERENCES

[1] Tadao Murata, "Petri Nets: Properties, Analysis and Applications, *"Proceedings of the IEEE*, Vol. 77 No. 4, pp.541-580, 1989 April .

[2] P. David Scotts and Richard Furuta, Petri-Net-Based Hypertext: Document Structure with Browsing Semantics, *ACM transactions on Information Systems, Vol. 7, No. 1*, pp 3-29, 1989, January.

[3] James L. Peterson, Petri Net Theory and the Modeling of Systems, Prentice-Hall, 1981.

[4] X. Li and W.B. Mugridge, Petri Net Based Graphical User Interface Specification Tool, *Proceedings of Software Engineering Education and Practice Conference*, IEEE CS Press, pp.50-57, 1994, Nov.

[5] X. Li and W.B. Mugridge, Extensions to the PUIST User Interface Specification Tool, IEEE CS Press, *Proceedings of Software Engineering Education and Practice Conference*, 1996, January.

[6] Xiaosong Li, Warwick B. Mugridge and John G. Hosking, A Petri Net-based Visual Language for Specifying GUIs accepted by IEEE VL'97

[7] P.A. Palanque, R, Bastide, L. Dourte, and C. Sibertin-Blanc, Design of user-driven interfaces using Petri nets and objects, *LNCS 685*, pp 569-585, 1993.

[8] R. Bastide and P.A. Palanque, A Petri net based environment for the design of event-driven interfaces, *LNCS 935*, pp 66-83, 1995.

[9] Y. Deng, S.K. Chang, J. Figueired and A. Perkusich, "Integrating Software Engineering Methods and Petri Nets for the Specification and Prototyping of Complex Information System", *Lecture Notes in Computer Science 691, Advances in Petri Nets*, 206-223, 1993.

[10] H. Oswald, R. Esser and R. Mattmann, "An Environment for Specifying and Executing Hierarchical Petri Nets", *Proc. of the 12th International Conference on Software Engineering*, pp. 164-172, 1990.