A Petri Net-based Visual Language for Specifying GUIs

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 language for graphical 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 and prototyping.

1. Introduction

PUIST (Petri net based graphical User Interface Specification Tool) is a visual language for specifying the static form and dynamic behaviour of graphical user interfaces. Petri nets [1] are used as the specification notation. Petri net nodes may be associated with GUI objects, such as windows and menu items. The PUIST environment also allows realisation and execution of the Petri net specifications, permitting prototype user interfaces to be tested concurrently with their specification.

A common problem with large Petri nets is the "spaghetti mess" of resulting interconnections that make them difficult to understand. Thus, despite benefits, such as their amenability to formal analysis [1,2], they have had little success as a large scale specification notation. PUIST solves this problem by introducing modularity into the specification language in the form of *subnets* which permit complex Petri nets to be specified in a generative form.

The paper commences with a brief description of the PUIST notation, followed by an example illustrating how large Petri nets arise. The subnet specification notation is then introduced, together with an example illustrating its use. A description of related work is followed by a discussion and conclusions.

2. Basic PUIST notation

PUIST's Petri nets are graphs consisting of two types of nodes, *places* (circles) and *transitions* (boxes), and *arcs* connecting places to transitions or vice-versa. Each place can have a nonnegative integer number of *tokens*. The right hand side of Fig.1 shows a Petri net with two places (p1, p2), three transitions (t1, t2, t3) and one token in place p1.

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 fires 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' of t, w(t,p') being the weight of the arc from t to p'.

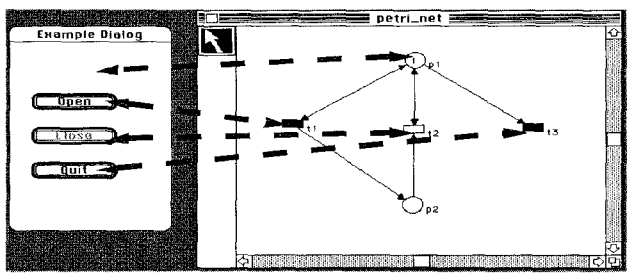

Fig.1. A dialog box with the corresponding Petri net

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.

For example, Fig. 1 shows a dialog (associated with place p1) with three buttons (associated with transitions t1, t2, and t3). A token in p1 means that the dialog box is open. Transitions t1 and t3 are enabled (as shown by their icons being filled in) because of the token in p1, thus enabling the "open" and "Quit" buttons. Place p2 is

1049-2615/97 $10.00 © 1997 IEEE

associated with a window. In Fig.1, it is empty so the window is hidden. If the "Open" button is clicked, transition t1 fires, removing the token from p1 and passing tokens back to p1 (due to the bidirectional arc) and on to place p2. This causes the window to open and the "Close" button to be enabled, as shown in Fig. 2. If the "Close" button is then selected, the net returns to its initial state.

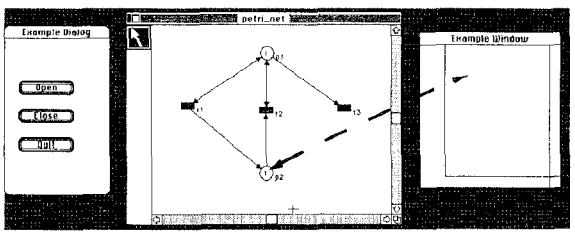

Fig.2. Dialog box of Fig.1 after the Open button is hit

Each transition may have an associated firing routine (a Prolog predicate), to be invoked upon firing the transition. Aspects of the underlying application semantics can thus be modelled to make the prototype more comprehensive.

To specify complicated dependencies among elements, PUIST provides three specialised components. *Emptying arcs*, are specialised place-to-transition arcs: the transition may be enabled independently of the number of tokens in the place. When the transition fires, the place becomes empty. *Inhibitor arcs* also connect a place to a transition: only when the place is empty can the transition be enabled. As long as an *auto-firing transition* is enabled it fires, i.e. a triggering event is not required. Further details of the basic PUIST notation may be found in [3, 4].

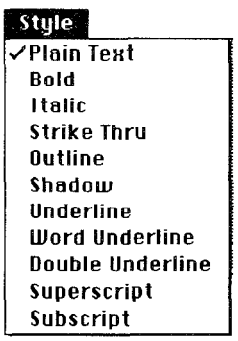

Fig.3. A menu with mutually exclusive and mutually compatible menu items

3. A Larger Example

Fig. 3 is a simplified MacWrite Style menu from [5]. Plain Text is the default, so it is ticked (ie selected) if no other item is ticked. Plain Text and the whole group {Bold, Italic, Strike Thru, Outline, Shadow, Underline, Word Underline, Double Underline, Superscript, Subscript} are mutually exclusive, so if any one in the group is ticked, Plain Text becomes unticked and vice versa. Bold, Italic, Strike Thru, Outline, Shadow, group {Underline, Word Underline, Double Underline} and group {Superscript, Subscript} are mutually compatible: any number of them can be ticked at the same time. Superscript and Subscript are mutually exclusive. In addition, when a ticked menu item (except Plain Text) is selected, it becomes unticked.

Fig. 4 shows the Petri net specification of this menu. In contrast to other approaches, such as Lean Cuisine [5], this specification not only describes the mutually exclusive and mutually compatible relationships among the menu items but also their dynamic behaviour. PUIST also generates a prototype implementation of the menu, which interacts with the Petri net during simulation. The Style place represents the menu Style. Transition B represents menu item Bold and Bp represents the tick status of Bold. When B fires, Bp gets a token indicating that Bold is to be ticked. B1 is an automatic firing transition with a weight two arc leading to it. A second selection of Bold puts a second token in Bp enabling B1 which immediately fires. This consumes all the tokens in Bp so Bold becomes unticked. The I, ST, O and Sh places and transitions similarly model the Italic, Strike Thru, Outline, and Shadow menu items.

Sup and Sub represents the mutually exclusive Superscript and Subscript menu items. Emptying arcs connect Supp to Sub and Subp to Sup specifying the mutual exclusivity: if one of Sup or Sub fires the ticking place for other is emptied. The U, W, D group of components similarly represent the mutually exclusive Underline Word Underline, and Double Underline items.

Finally, a significant amount of "plumbing" is associated with implementing the Plain Text default. The Plain Text item is represented by transition P, and its tick status by place Pp. Emptying arcs from place P to the other style transitions guarantee Plain Text is unticked if any of those transitions fire. Similarly, emptying arcs from the other style tick status places to transition P ensure no other style is ticked if Plain Text is selected. Inhibitor arcs from each tick status place connect to the Default automatic firing transition, ensuring it is fired when ALL items are unticked, setting Plain Text to be ticked.

While Fig. 4 captures the static and dynamic requirements of the menu, it is too complex to easily understand, markedly reducing its value as a specification. It is clear there are repeating patterns and some hierarchical organisation, but the complexity primarily arises from the large numbers of interconnections required for the mutual exclusivity and default behaviour. In the next section we describe *subnet* specifications which permit such complex, yet regular, Petri nets to be specified simply.

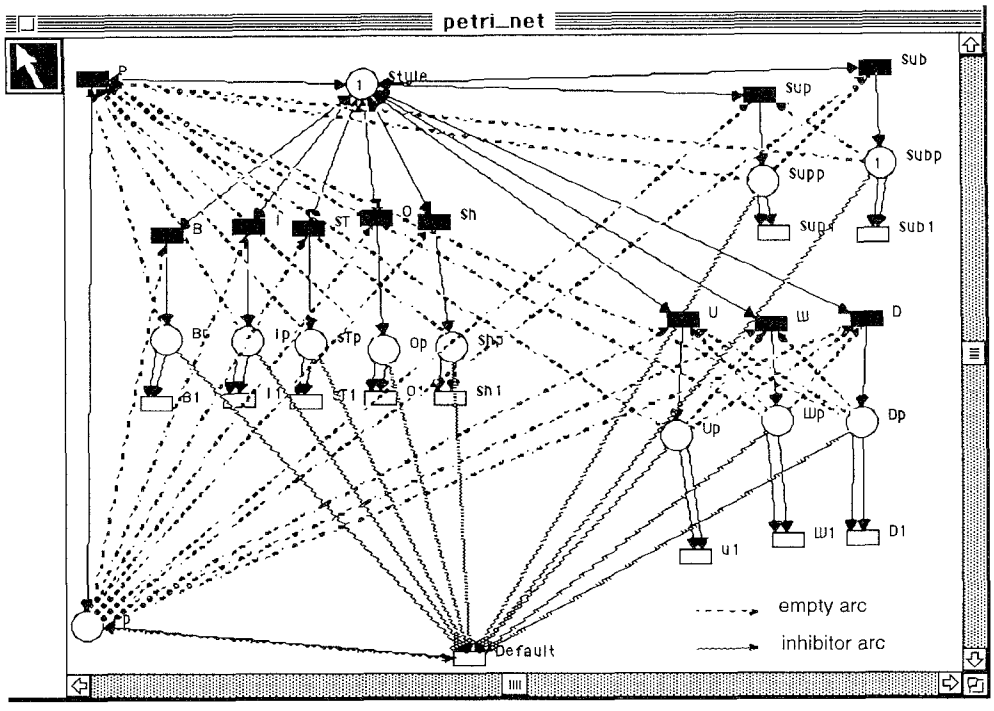

Fig.4. The Petri net for the menu in Fig.3

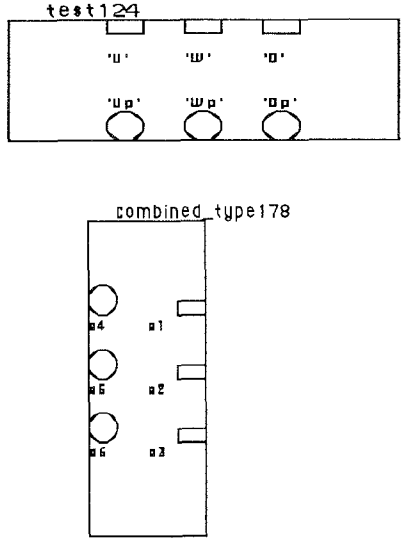

Fig.5 Example subnet icons

4. Subnets

A subnet type specification defines an abstract Petri net structure which can include recursive generative components and can produce different Petri nets with different GUI semantics dependent on specific instantiation parameters. Subnets are instantiations of a subnet type specification. Visually, a subnet is represented as a rectangular icon with named connection points, representing internal places and transitions accessible from outside the subnet, along its edges, as shown in Fig. 5.

4.1 Subnet type specifications

Fig. 6 shows a subnet type specification with two formal parameters. This defines subnets that consist of a set of mutually exclusive menu items, such as the Super/Subscript or Underlining groups of the Style menu. Type specifications may include a number of *cases*. Each case is differentiated by its parameter list, specified in a box at the top, and its definition inside the window area. Prolog-style pattern matching is used to select between the cases when a specification is instantiated. In Fig. 6, the first case, exclusive_items[1], has two parameters. The first is the variable MenuName, while the second matches an empty list. In the second case, exclusive_items[2], the first parameter is the menu name variable, while the second matches the head and tail of a non-empty list, each element being a menu item name.

The internal specification of each case can contain standard PUIST elements (places, transitions, and arcs) together with applications of subnet type specifications, *instantiation bindings*, and *connection specifiers*. The latter

specify the external connections resulting from the subnet, i.e. which internal components are connectable from outside the subnet, and their relative positions on the edges of the subnet icon. In the example in Fig. 6, there are two connection specifiers for each case: a list of transitions (transition collection) on the top edge, and a list of places (place collection) 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. In exclusive_items[1], there are no such "wires", so both lists are empty (ie there will be no transitions or places exported from the instantiation of exclusive_items[1]).

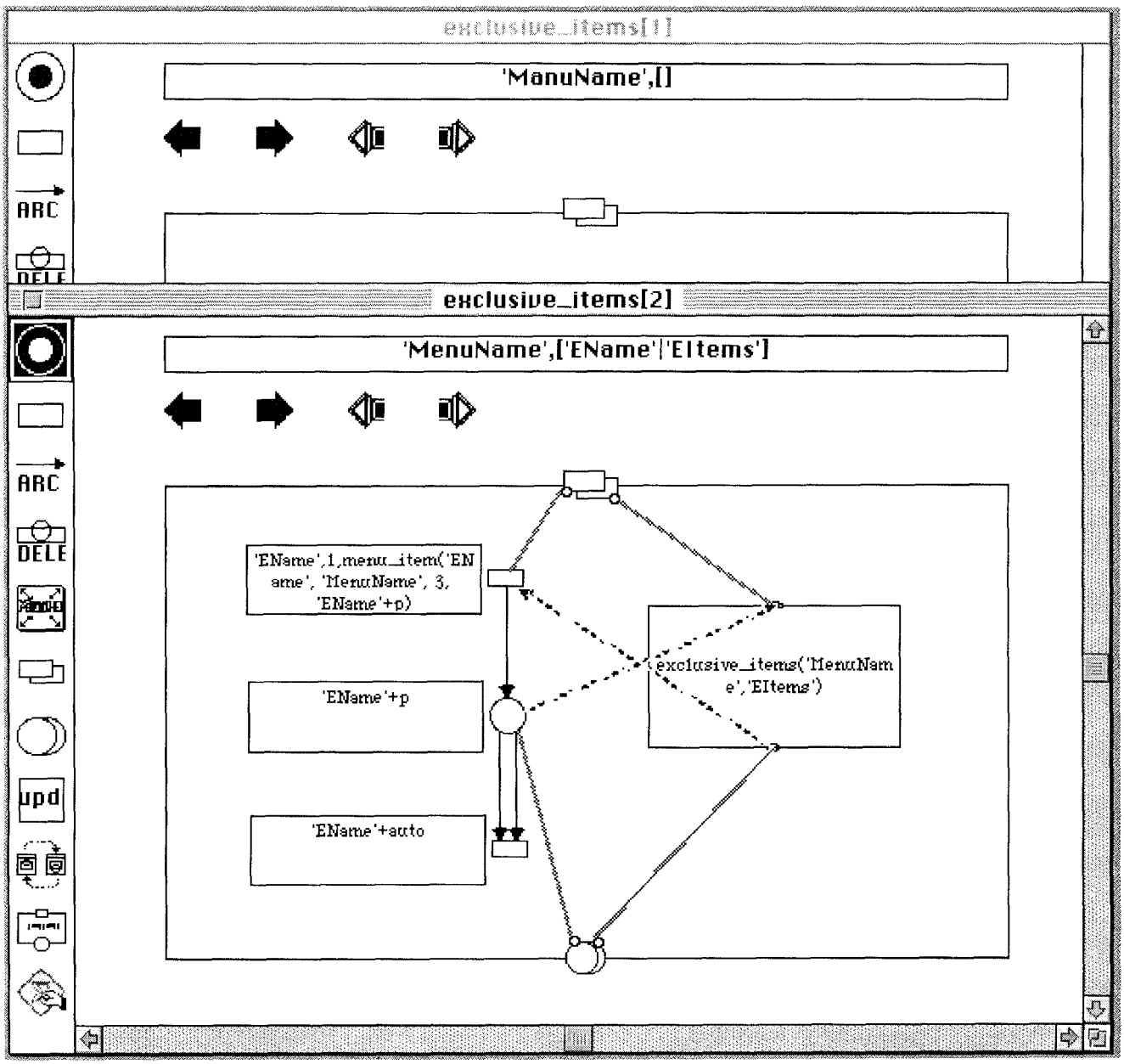

Fig. 6: Subnet type specification for exclusive items

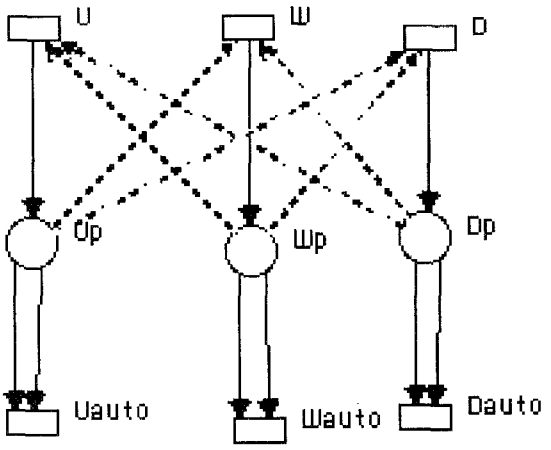

Fig.7 The underline group subnet instantiated from the subnet type of Fig.6

In the exclusive_items[2] case, two transitions and a place are connected together to represent one element in the mutually exclusive set. *Instantiation bindings* (▭) are used to name and initialise the internal transitions and places. A variety of parameter binding options are provided for a transition, such as: its name; whether it is autofiring; whether a GUI object is associated with it and, if so, its type and instantiation parameters. Places have similar binding options. For example the top transition is defined to be a menu_item GUI object with name EName, associated with menu MenuName (and other parameters to initialise the menu item). The names of the middle place, representing the ticking or selection status of the menu_item, and the bottom (auto-firing) transition are defined to be the menu item name with an additional "p" and "auto" appended respectively. On the right, there is a recursive application of the exclusive_items Petri net type with the same menu name and the tail of the menu item name list as arguments. The exclusive_items[1] case acts as the base case of the recursive specification.

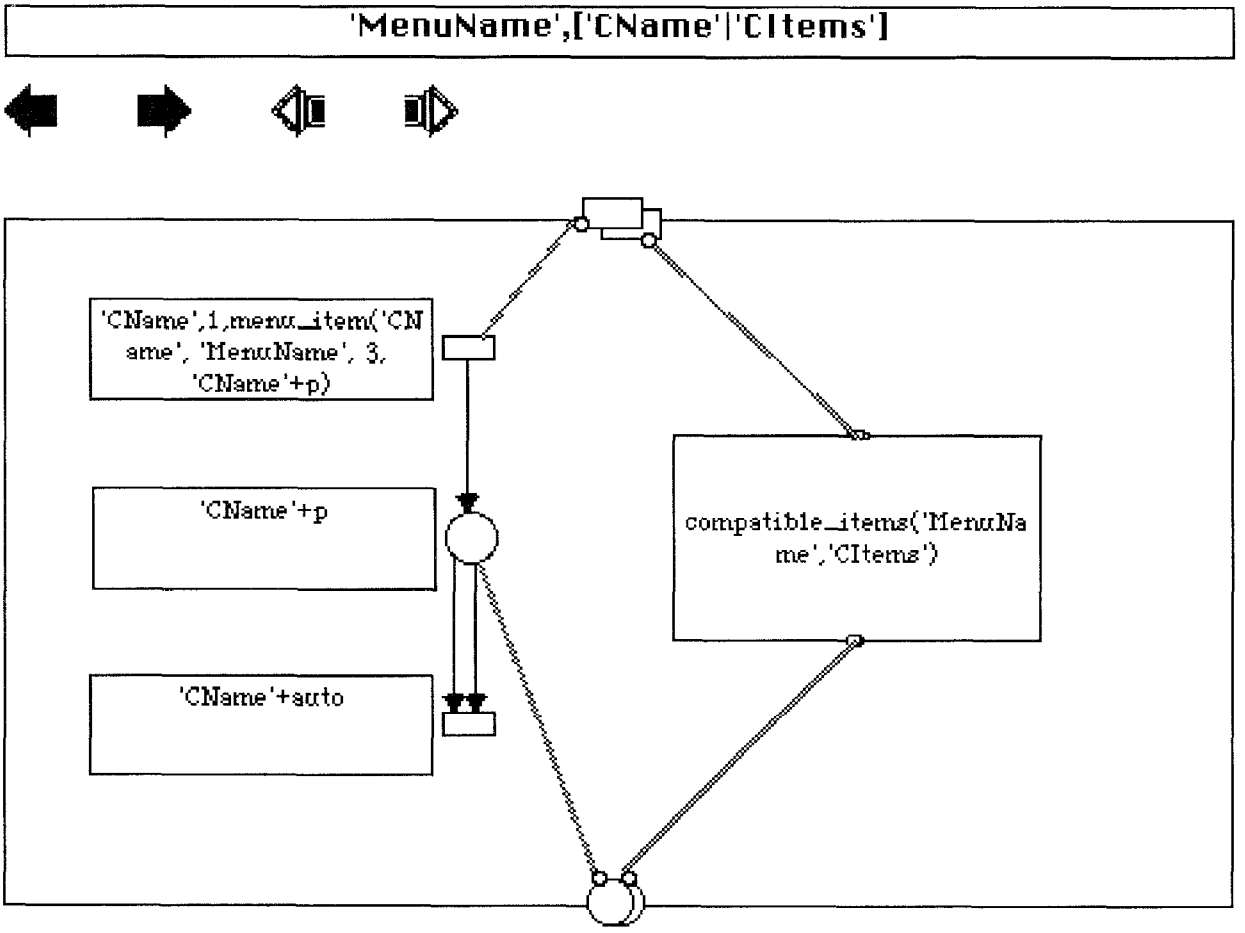

Fig. 8: Recursive case of compatible_items specification

The output connection wiring (grey) prepends (indicated by the ordering of the "pin" connections) the menu item transition to the top list of menu item transitions exported from the recursive application. Similarly it prepends the ticking status place to the recursively exported list of ticking status places at the bottom of the subnet type. Internally, an emptying arc connects the menu item transition to *each* of the places, that are exported from the recursive instantiation. Likewise, *each* of the exported transitions is connected by an emptying arc to the ticking status place. These single lines thus represent collections of arcs, simplifying the complex interconnections visible in Fig. 5. The arcs ensure only one of the generated elements has its ticking status set.

4.2 Subnet instantiation

A subnet is instantiated by supplying actual parameter values to a Petri net type specification via a dialog box. Fig. 7 shows the Petri net resulting from an instantiation of exclusive_items with three entries in the menu item list (U,W,D). This net corresponds to the underline mutually exclusive menu item group of the style menu example. The instantiation can also be viewed as a subnet icon (top subnet icon in Fig. 5), which hides the internal wiring of the generated Petri net and just presents the exported transitions and places in the order specified by the recursive generation across the top and bottom respectively of the subnet icon. These can then be "wired" to as if they are any normal transition or place.

5. Specification of the Style Menu

The exclusive_items type can be used as one component of a complete specification for the Style menu example of Fig. 3. In addition, a specification for the mutually compatible components is needed, together with the "plumbing" to manage the default Plain case.

Fig. 8 shows the recursive case of a specification for a compatible items subnet type. This follows a similar form to the mutually exclusive case, but omits the emptying arcs, allowing multiple ticking status places to hold tokens at once. The base case (not shown) is empty. Fig. 9 (left) shows the style_items subnet type which specifies all of the style menu items, except the Plain Text item and its default processing. Style_items is parameterised by three menu item lists corresponding to the "standard" styles, the underline group, and the super/subscript group. Constant bindings could just as easily have been used in this specification, avoiding the parameterisation. Exported are a list of the menu item transitions and ticking status places of all of the composite menu items.

Figure 9 (right) shows the whole style menu specification, including the Plain Text menu item, and the default processing connections. The structure of the menu is now quite clear: the emptying arcs have a similar role to those in the mutually exclusive items specification, with the inhibitor arcs associated with the Default automatic firing transition, ensuring the Plain Text is selected if none of the others are. The only component exported is the place representing the Style menu as a whole. Again, the specification has been parameterised by the menu item data, although constant lists could just as easily have been used. Fig.10 shows an automatically generated subnet which results from instantiating the style menu type of Fig. 9 (with appropriate menu item names, etc) together with the prototype GUI menu component. The resulting Petri net is live and hence the prototype menu can be tested dynamically.

6. Related Work

A number of 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 Colored Petri nets, using typed tokens for comunication. However, these systems lack any form of modularity such as the subnet types of PUIST. Other groups have been investigating formal properties that can be extracted from analysis of computer systems specified by Petri nets [1,2].

The work most closely related to the Petri net type specification mechanism comes from another area: hardware design specification. Hardware design involves specification of wiring between complex, but often repeated, elements, a problem very similar to that addressed by PUIST. Smedley [8], for example, describes a visual language for specifying digital circuits which uses a modular/recursive wiring approach similar to that of PUIST. Smedley's system is itself implemented using Prograph [9], a visual dataflow programming language which also has modular/recursive structuring similar to that of PUIST, but with dataflow rather than Petri-net token flow semantics. Recently Cox and Smedley have also applied similar techniques to specification of structured graphical objects [10].

7. Discussion and Conclusions

We have presented and implemented a visual language for specifying GUIs using a 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.

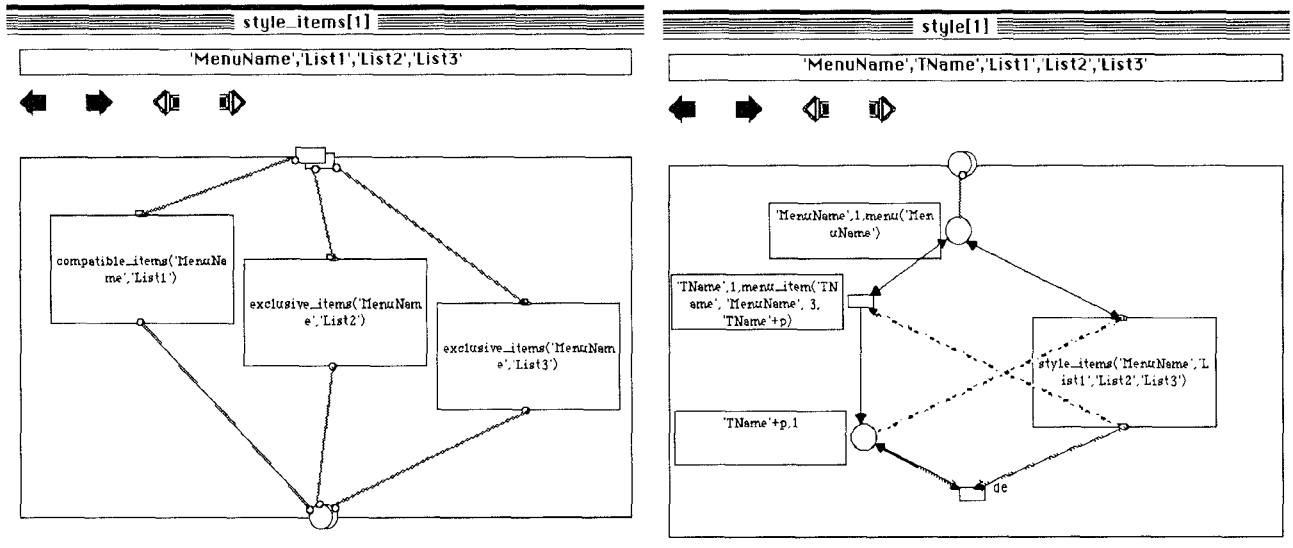

Fig. 9: style_items subnet type specification (left) and whole style menu subnet type specification (right)

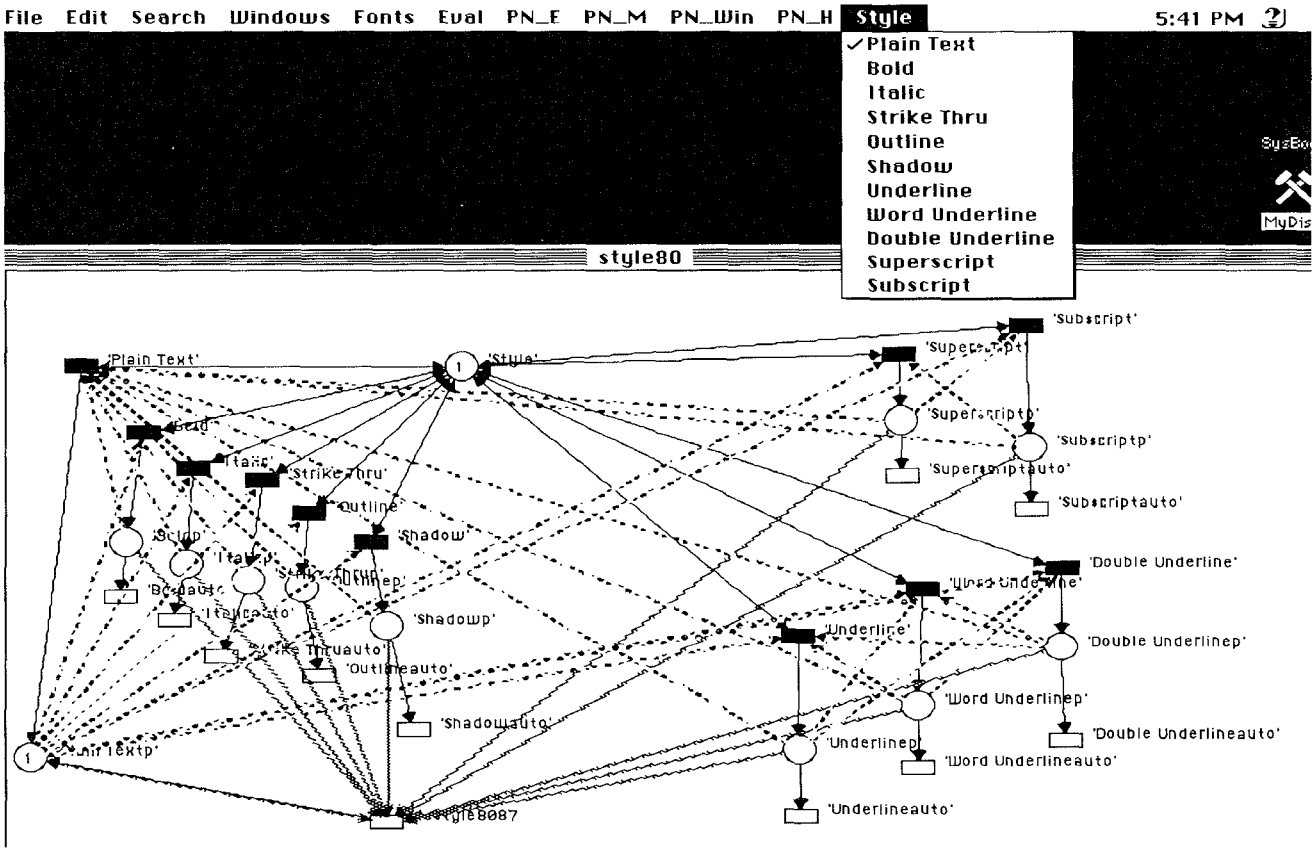

Fig. 10: An instantiated style menu specification

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. The visual notations used for subnet specification use a similar style to Prograph [9] method definitions in their use of attachment points and different flavours of "flow" wire, but differ significantly in the underlying semantics represented (Petri net token flow rather than dataflow).

While the language has a simple notation, it is capable of defining complex Petri nets. It is particularly good for Petri nets with complex and repeated pattern structures, where abstractions can be applied.

We are currently constructing a library of standard subnet types, including the exclusive and compatible items 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], to the analysis of GUIs specified using PUIST. We expect to extend such analysis to the evaluation of specified GUIs according to Petri net properties, with visualization of the evaluation metrics.

Other possibilities for the application of PUIST's subnet formalism are also being investigated. These include the provision of modular specifications for Petri-net based software process specifications, such as SLANG [11] and ProcessWEAVER [12]

References

[1] W. Reisig, Combining Petri nets and other formal methods, *LNCS 616*, pp24-39, 1992.

[2] C. Sibertin-Blanc, Cooperative nets, *LNCS 815*, p471-490, 1994.

[3] 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, November.

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

[5] M.D. Apperley and R. Spence, "Lean Cuisine: a low fat notation for menus", *Interacting with Computers*, Vol 1 No 1, pp. 43-68, 1989.

[6] 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.

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

[8] T.J. Smedley, A high-level language for the graphical description of digital circuits, *Proc VL'95*, IEEE CS Press, pp77-82, 1995

[9] P.T. Cox, F.R. Giles and T. Pietrzykowski, "Prograph: A step towards liberating programming from textual conditioning", *IEEE Workshop on Visual Languages*, pp.150-156, October1989.

[10] P.T. Cox and T.J. Smedley, A visual language for the design of structured graphical objects, *Proc VL'96*, IEEE CS Press, pp296-303, 1996.

[11] Bandinelli, S., Fuggetta, A., Ghezzi, C., and Lavazza, L., *SPADE: an environment for software process analysis, design and enactment*, Software Process Modelling & Technology. Finkelstein, A. and Kramer, J. and Nuseibeh, B. Eds, Research Studies Press, 1994.

[12] Fernström, C., "ProcessWEAVER: Adding process support to UNIX," in *2nd International Conference on the Software Process: Continuous Software Process Improvement*, IEEE CS Press, Berlin, Germany, February 1993, pp. 12-26.