A High Level Language for Specifying Graph Based Languages and their Programming Environments (Draft)

****inc fixes of 920922****
****inc fixes of 930912****

M.F. Kleyn and J.C. Browne
kleyn@cs.utexas.edu browne@cs.utexas.edu

Abstract

This paper describes a high level language for specifying programming environments for programming languages that are based on directed attributed graphs. The high level language allows the specifier to describe views of portions of a program written in such a graph-based language, the editing operations used to create the program, animations of the execution of the program, and sufficient detail of the execution semantics to support the animations. We demonstrate the use of the specification language with two simple examples of graph-based languages: Petri Nets, and an extension of Petri Nets which includes the ability to nest nets hierarchically. We further describe how to generate the programming environment for graph-based languages from descriptions made in the specification language. This work is the basis for developing a compiler for generating programming environments for graph-based languages automatically. We wish to remedy the add-hoc re-inventing of such systems by providing the high-level domain-specific set of abstractions for specifying them. The specification language is based on using a grammar to describe the components of the graph-based language and using a first-order logic based language to describe state changes in editing, execution, and animation.

1.0 Introduction

As the use of graphical user interfaces (GUIs) becomes more prevalent, so is the use of graphical notations for writing programs. Though *graphical* programming systems vary widely, a great many, if not most, are based on the manipulation of an underlying *graph* structure [EGF91][1]. This paper proposes a means of formally describing such systems. We collectively term these programming languages "Graph-Based Languages" (GBLs), and refer to the languages together with their programming environment as "Graph-Based Language and Environments" (GBLEs). In GBLs, programs are defined by interconnecting different types of nodes and links, and by filling in values of attributes associated with the nodes and links in the graph. The class of graph used is usually a directed graph, but it may be more general, such as a multigraph or a hypergraph. GBLs are in very widespread use, across many application domains. The following sample of existing GBLs shows the wide variety of domains in which they are used:

1. We thus make a strong distinction between the a *graphical* object - something that is displayed that does not involve text, and a *graph* object - a mathematical abstraction that can be displayed graphically.

- State transition systems in which the states are represented as nodes and the transition between them as edges. The system in[Jac85] is a state transition graph system for specifying dialogs in user interfaces. The PROSPEC uses coupled pairs of state transition graphs for specifying and validating network protocols [CL88].
- Commercial GBLs for specifying image and signal processing, and graphics rendering such as AVS [Sta91], IDF [Hun90], and Explorer [Sil92]. These systems are typically used to specify the interconnection of modules written in standard textual programming languages.
- Data-flow visual programming systems such as those surveyed in [Hil92]. These are similar to the systems in (2), except that a graph in these languages defines the computation at a finer grain. These systems have an underlying data-flow model of computation but are not directly intended for parallel programming.
- Parallel programming GBLs. The large number of GBLs for parallel programming can be seen in the surveys of such tools [Che91] [CS90] [LER92] [FJ86]. Recent examples include ParaGraph [BCL90], Darwin [MK92], Enterprise [SSW+92], HeNCE [BDG+91], Phred [BN91], and CODE2 [NB92].
- Environments for specifying real-time systems such as Modechart [JM88]. Modechart is derived from Statecharts [Har86] - a hypergraph based GBL for specifying reactive systems.

While using pen-and-paper or a text-editor can be sufficient for writing a textual programming language, GBLs *need* a programming environment to be used effectively. Building an environment that is easy to use and clearly reflects the features of a new GBL is a tedious and complex task. This work addresses this problem by providing a concise and precise way of describing such programming environments through a declarative high level (textual) language. A programming environment for a new GBL can then be generated automatically from this declarative description. Systems which automatically create programming environments for textual languages have been developed, notably Gandalf and the Cornell Program Synthesizer[ea85] [Rep84]. However, no general means for specifying GBLEs which uniformly integrates graph and text manipulation exists, despite their widespread use.

This work differs from *graphical* user interface libraries such as *Interviews* [MHC89], *Unidraw* [VL89], and *Grow* [Bar86], since these are not specifically targeted at graph interfaces. It is also an advance on graph display, animation, and manipulation libraries such as *Edge* [PT90], *TGE* [KS90], *XGRAB* [RDM$^+$87], *Xsim* [Tho90], *GMB* [JG89], and *DAG* [GNV88]. These libraries or object toolkits provide the low level support for developing graph based applications but they are general purpose systems and not targeted to any particular domain. Our work addresses the synthesis of programming environments. We are raising the level of abstraction further; our compiler targets these libraries. We aim to show the value of high-level languages in this real example. It is by narrowing the scope of our specification language to the domain of GBLEs that we make automatic generation problem tractable. GBLEs provide a challenge for high level languages because of their complexity, but therefore will also a provide a significant payback.

This work is also innovative in including program language semantics and program animation. Animation is an increasingly important component of programming environments. The animation of graphical displays of programs can be of great assistance in understanding program execution and locating the source of incorrect program behavior, especially in parallel programs[HC88] [KG88]. We wish to be able to specify simple animations, based on the semantics of the GBL, and targeted towards assisting the program developer with correctness and debugging. We allow the specifier to describe, at some level abstraction, the execution behavior of the program. This description of execution semantics need not describe execution in complete detail; only enough to capture behavior required for the animation. For example, some parallel GBLs allow the programmer to associate a program written in a standard sequential language with a node. In this case the amount or lack of detail about the behavior of the sequential program that is included is at the discretion of the GBLE specifier. The animations can then be described by tieing them to this abstracted description of program behavior - they are intended to reflect the run time activity of the program. This form of animation differs from the more pedagogical animation systems that have been developed to illustrate how particular well-known algorithms work [Bro88] [Sta90]

In developing our specification language, we aim to encompass GBLEs for parallel as well as sequential programming. Parallel GBLs are useful for "explicit" parallel programming in which the parallelism is specified by the programmer, rather than extracted by a compiler. Graph displays in parallel GBLEs are helpful in conveying the structure of multiple threads of control that exist in a (MIMD) parallel program. Browne et al. have shown how such parallel models of computation can be consolidated under a unified model of computation [Bro85]. The CODE system derived from this unified model of computation, now in its second generation, is one of the more sophisticated GBLs for parallel programming [NB92]. It uses a set of structured attributes associated with several types of nodes and links. It is hierarchical as are many parallel programming GBLs, and uses graph diagrams as templates which are dynamically instantiated at run-time. This work was initiated in the context of CODE and it is our goal to formally specify the GBLE for large systems such as CODE, though this paper focuses on explaining the principles of our approach with simpler Petri Net GBLEs.

A realized GBLE would typically have the following appearance: The system would have a main window displaying a (top-level) graph containing a set of nodes and links whose appearance reflects some important property (*e.g.*, the node type). Attributes of nodes and links would be made available for editing by a pop-up form window. The form would appear when a node is selected. A side-bar menu would make available commands for displaying new windows, invoking graph manipulation editing operations, initiating the execution of the program, and selecting a desired animation.

The proposed language covers the specification of:

- *Views* - Views correspond the contents of windows that show parts or collections of parts of a GBL program.
- *Editing* operations - Editing operations are the programmer-initiated operations that are used to create and modify a program.
- *Execution* behavior of the program. The underlying model of computation associated with the GBL, can be specified. This is done by formally specifying the semantics of the GBL, or an abstraction sufficient to support the desired level of animation.
- *Animation* - Animations are changes to views that show how the program is executing.

This paper is organized as follows: Section 2 describes the specification language for describing GBLEs, "GLIDE". Expressions used in the specification of a GBLE for a simple form of Petri Nets are used to illustrate the main concepts. In Section 3, the complete GLIDE specifications for a more complex Petri Net language, PCM, are given. The latter is an extension in which nets can be hierarchically nested and nets can be dynamically created at run-time. In Section 4 we describe the algorithm used to compile the specifications. The appendix (Section 8) provides the full formal specification of the GLIDE syntax.

1.1 Definition of Terms

Before describing our specification language it is useful to review and define more precisely the terminology mentioned in the introduction and used throughout this paper:

Programming Environment - A system which allows a programmer to create, execute and animate programs of a given programming or specification language. The term *Software Development Environment* is also often used.

GBL - Graph-Based Language - any programming or specification language that uses a directed graph as basic representation medium. The graph is usually explicitly displayed, but this need not be the case.

GBLE - Graph-Based Language and Environment - the language *and* its programming environment.

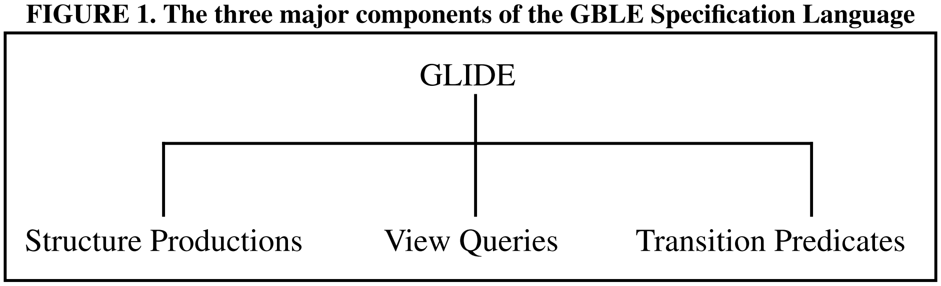
ponent, Attribute - Graph-Based Languages have nodes and links which have attributes (*e.g.*, a firing rule). These attributes may be structured so they themselves have attributes (*e.g.*, rule-lhs and rule-rhs). The term component is used to refer to all of these items (nodes, links, rules, rule-lhs, etc.) generically.

Action, Events, Transitions, *Operations* - These terms all refer to entities associated with change of state. The term *event* indicates a particular instance of an *action*. The term *transition* is used to emphasize that an event or action is atomic. *Operations* are actions that can be externally invoked (*e.g.,* an editing operation invoked by the programmer).

2.0 Specification Language

This section describes GLIDE, the formal language for specifying a GBLE. A specification in GLIDE consists of three parts: (i) a set of productions for specifying the structure of a GBLE, (ii) a simple query language for identifying collections of components of a GBLE program for inclusion in a view, and (iii) collections of predicates for describing transitions and transition invariants of the GBLE (Figure 1):

FIGURE 1. The three major components of the GBLE Specification Language



The structure productions are written in an extended BNF, the view queries are expressions which define sets, and the transition predicates are in a predicate calculus based language.[1] Our approach to the specification of the structure of GBLs is thus modeled after the way textual languages are normally specified. However, many concepts used in the formal specification of textual programming languages must be extended and adapted to accommodate some of the non-textual characteristics of GBLs and the fact that we are specifying the language's programming environment as well. We now describe each of these three parts of GLIDE in more detail, using the specification of Simple Petri Nets (SPN) GBLE as example.

2.0.1 Structure Productions

The specification of the structure of a GBLE takes the form of a set productions which describe the components of the graph and how they are combined. For example, in describing a Petri Net GBLE, a specifier might wish to state that it consists of a *set* of nodes *and* a *set* of links and that the nodes can be *either* place nodes *or* transition nodes. This informal description is captured in the three productions:

PetriNet == *N* *: PNNode**;*
L *: PNLink***

PNNode == *P* *: PlaceNode* |
T *: TransNode*

1. The notation borrows, as much as possible, from the Metanot notation used by Meyer in [Mey90] to specify the syntax and semantics of programming languages.

The annotations used in grammar are summarized in the following table:

Table 1: Grammar Symbols

Connectors between Components			*Component postfix Annotations*		
Symbol	Name	Meaning	Symbol	Name	Meaning
;	*semi-colon*	*aggregation*		*no annotation*	*single element*
|	*vertical-bar*	*choice*	*	*asterisk*	*sequence*
	space	*concatenation*	**	*double asterisk*	*set*
			***	*triple asterisk*	*bag*

The labels *PN, N*, *L*, *P*, and *T* are optional *tags*. Tags provide unique names for components of a production and are later used to identify these components in the view query and in the transition predicate expressions. The node type *PlaceNode* can be further decomposed a production.

PlaceNode	==	*Id*	*: INTEGER;*	
		NmTk	*: INTEGER;*	*/* number of tokens in a place node */*
		I	*: PNLink**;*	
		O	*: PNLink**;*	
		Gr	*: Graphic*	

Uppercase is used to indicate a primitive type (*e.g.*, *INTEGER*), or the values of an enumerated type. The *I* and *O* components of this production associate a set of input and a set of output links with place nodes.

A similar production defines transition nodes.

TransNode	==	*Id*	*: INTEGER;*
		Ste	*: State;*
		I	*: PNLink**;*
		O	*: PNLink**;*
		Gr	*: Graphic*

The annotations define the *class* of the graph. Standard Petri Nets are of the class multigraph (a place node and transition node may have many links between them), and therefore in the definition of standard Petri Nets, *I* and *O* would be *bags*. However, we are describing a simpler Petri Net model in which there is at most one link between a place node and transition node. Therefore, we restrict the values of *I* and *O* through the definition of *PNLink.*

PNLink	==	*Lp*	*: PlaceNode;*
		Lt	*: TransNode;*
		Gr	*: Graphic*

Some of the attributes of the nodes and links are further decomposed.

State	==		*: ENABLED | FIRING | DISABLED*
Graphic	==	*S*	*: Shape;*
		Lbl	*: Label;*
		Hl	*: Highlight;*
Shape	==	*R*	*: Radius;*
		T	*: Thickness*
Label	==	*Ts*	*: TextString;*
		Fn	*: Font*

Flash : Function / animation routine */*

The last three productions describe attributes associated with the programming environment, not the Petri Net Language per-se. A font attribute is not required to define the semantics of the Petri Nets, but the attribute does affect how the Petri Net is displayed. We are thus coupling the graphical attributes tightly with the language. The syntax of GLIDE productions is given in the Appendix (Section 5.1).

2.0.2 View Queries

A typical GBLE will usually have a main display window showing how the nodes and links are connected. It may also have subwindows for editing selected attributes of a node or link. In order to specify the contents of such windows we regard the productions for the GBLE as a database schema, and a program as the data in the database. We term the result of queries into the database *views*. The view query syntax is based on identifying components in the productions. Continuing with the Petri Net example, a view showing the set of all the nodes and all the links is specified by the expression:

$$WholeGraph(pn{:}PetriNet) == \{x \mid x \in pn.N.P \vee x \in pn.N.T \vee x \in pn.L\}$$

The parameter expression *pn:PetriNet* indicates that *pn* is a particular instance of a Petri Net. A "dot-addressing" notation is used to specify a subcomponent of a component, thus *pn.N.P* denotes the set of all place nodes of *pn*. A display of only the place and transition nodes is specified by the query

$$Place\&TransNodes(pn{:}PetriNet) == \{x \mid x \in pn.N.P \vee x \in pn.N.T\},$$

whereas the display of only the place nodes with their labels is specified by the query:

$$PlaceNodes\&Labels(pn{:}PetriNet) == \{x, x.Gr.Lbl \mid x \in pn.N.P\}.$$

Finally, the following query shows the label and the number of tokens of a particular place node.

$$ShowAllNumTokens(p{:}PlaceNode) == \{ x.Gr.Lbl, x.NmTk \mid x = p \}$$

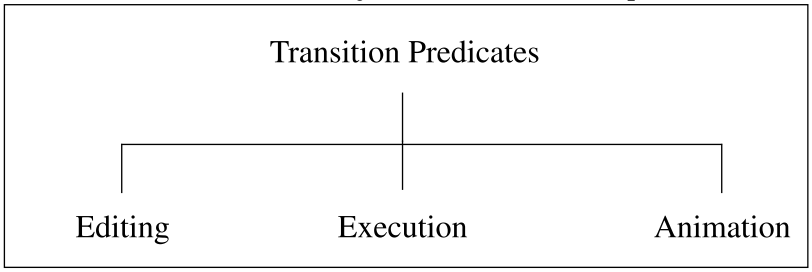
The specification of views differentiates a GBLE from text editors and textual languages. A view corresponds to displaying a part of a textual program (*e.g.*, a segment of a text file or a subroutine definition), but it is a more general concept. The query language allows the specifier to state what components to show, and in how much detail to show them. Displaying a non-terminal component of a production is a means of showing a program at a lower level of detail, a higher level of abstraction. This is a further difference between using a GBLE and text editing since non-terminals are not displayed in most text editors. Some syntax-directed text editors do display non-terminals; the non-terminals are usually in a different font to indicate that they must still be replaced with a sequence of terminals as in, for example, [TR81].

View specifications are limited to specifying *what* components of a GBL program appear in a window but do not describe how the components are laid out in a window. The way in which the set of components identified by the query are eventually arranged in a window is not of our concern, we view layout as details to be addressed elsewhere (determined, for example, by the capabilities of the underlying GUI library).
The complete query syntax for view queries is given in the Appendix (Section 5.2).

2.0.3 Transition Predicates

The third aspect of GLIDE is the set of predicates. Transition predicates describe the semantics of several classes of operation that alter the components defined by the productions. There are three major categories of predicates as shown in Figure 2.

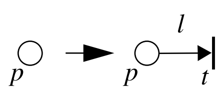
FIGURE 2. The three major classes of transition predicates



Editing transition predicates describe operations that change the structure of a program. Examples of editing operations include adding or deleting nodes or links, changing the interconnection in the graph, and changing the value of one of the attributes of the nodes or links. *Execution transition predicates* describe possible changes of program state during execution. This may involve movement of data along links, enabling nodes, or, in the case of GBLs that have dynamic run-time graphs, manipulation of the graph structure. *Animation transition predicates* couple changes in state associated with execution transitions to changes in the values of variables that represent graphical attributes, *e.g.*, fonts, highlighting, line thickness, etc. We now illustrate these three categories of transition predicate with examples from the Petri Net GBLE.

2.0.3.1 Editing

The following is an example of a predicate describing an editing operation.

	*AddTrans(p:PlaceNode) ==*	$l = new(PNLink) \wedge t = new(TransNode) \wedge p.O' = p.O \cup \{l\} \wedge t.I' = \{l\}$

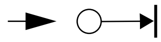
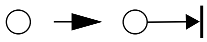
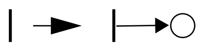
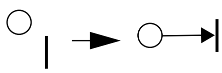
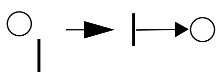
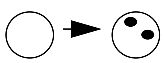
The definition of *AddTrans* specifies that an existing graph can be extended by adding a link (*l*) and a transition node (*t*) and attaching the link between an existing place node (*p*) and the new transition node. *new(X)* is a function that returns an instance of a type identified in the productions by the name *X*. The curly brackets, *("{}")*, are used to enclose a set. The expression $p.O' = p.O \cup \{l\}$ states that the *O* component of place node *p* contains the new element *l* after the invocation of this editing operation. We are using the widespread prime, *(" ′ ")*, notation to distinguish between the state of a component before and after a transition.[1] The expression *p:PlaceNode* indicate that p is and instance of the type *PlaceNode*. The predicate is a declarative description of the operation - it prescribes the relationship that must hold between values of components before and after the operation has been invoked.

AddTrans is a *syntax-directed* editing operation, since it generates a new syntactically correct Petri Net graph from an existing one (examples of a syntactically incorrect Petri Nets: a link between two place nodes, a link between two transition nodes, or a link that is unattached at one end). Editing operations can be specified to express both syntax-directed editing and less restricted forms of editing. In contrast to earlier syntax-directed editing systems, we have de-coupled the syntax of the language from the operations used to create an instance in the language. This flexibility overcomes the problem that pure syntax direct editors can be difficult to use[ea85] [FA90].

1. *e.g.*,[Spi88] [Lam90]

The following table is a full set of syntax-directed editing operations. They are sound and complete in the

Table 2: SPN Editing - Syntax Directed

	*Initial* ==	$[\ p = new(PlaceNode) \wedge$ $l = new\ (PNLink) \wedge$ $t = new(TransNode) \wedge$ $p.O' = \{l\} \wedge t.I' = \{l\}\ ]$
	*AddTrans(p:PlaceNode)* ==	$[\ l = new(PNLink) \wedge$ $t = new(TransNode) \wedge$ $p.O' = p.O \cup \{l\} \wedge t.I' = \{l\}\ ]$
	*AddPlace(t:TransNode)* ==	$[\ l = new(PNLink) \wedge$ $p = new(PlaceNode) \wedge$ $t.O' = t.O \cup \{l\} \wedge p.I' = \{l\}\ ]$
	*AddOutLink(p:PlaceNode, t:TransNode)* ==	$[\ l = new(PNLink)$ $p.O' = p.O \cup \{l\} \wedge t.I' = t.I \cup \{l\}\ ]$
	*AddInLink(p:PlaceNode, t:TransNode)* ==	$[\ l = new(PNLink) \wedge$ $t.O' = t.O \cup \{l\} \wedge p.I' = p.I \cup \{l\}\ ]$
	*SetNmTokens(p:PlaceNode, n:INTEGER)* ==	$[\ p.NmTk' = n\ ]$

sense that no sequence of such operations can create a syntactically incorrect Petri Net and any syntactically correct Petri Net can be created by at least one sequence of operations. The first operation, *Initial*, creates the minimal Petri Net; it has just one place node, one transition node, and one link between them. The last predicate, *SetNmTokens*, describes an initialization operation for Petri Nets. It allows the user to specify the initial number of tokens for a given place node. It is an initialization operation because it assigns a value to an attribute which is also modified by execution. Matching deletion operations are omitted for brevity.

For comparison, the following table shows the specification of editing operations which are not syntax-directed:

Table 3: SPN Editing - Not Syntax Directed

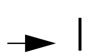
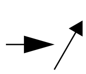
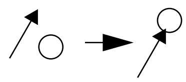
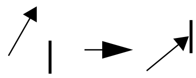
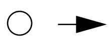
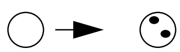
	*AddPlace(pn:PetriNet)* ==	$[\ pn.N.P' = pn.N.P \cup \{new(PlaceNode)\}]$
	*AddTransition(pn:PetriNet)* ==	$[\ pn.N.T' = pn.N.T \cup \{new(TransitionNode)\}\ ]$
	*AddLink(pn:PetriNet)* ==	$[\ pn.L' = pn.L \cup \{new(PNLink)\}\ ]$

Table 3: SPN Editing - Not Syntax Directed

	*LinkPlace(l:PNLink, p:PlaceNode) ==*	$[\ p.I' = p.I \cup \{l\} \wedge l.Lp = p]$
	*LinkTrans(l:PNLink, t:TransNode) ==*	$[\ t.I' = t.I \cup \{l\} \wedge l.Lt = t]$
	*DeletePlace(p:PlaceNode, pn:PetriNet) ==*	$[\ pn.N.P = pn.N.P' \cup \{p\}\ ]$
	*SetNmTokens(p:PlaceNode, n:INTEGER) ==*	$[\ p.NmTk' = n\ ]$

2.0.3.2 Execution

The same predicate notation is used to define the execution semantics of a GBL. Execution predicates describe changes of state that occur when a program executes. For example, the following predicate specifies that transition nodes are enabled if all the places connected to them by incoming links have at least one token:

$$Enable(pn{:}PetriNet) == \forall t{:}pn.N.T\ [\ \forall p{:}t.I.Lp\ [\ p.NmTk \geq 1\ ] \Rightarrow t.Ste' = ENABLED\ ]$$

The dot-address, *t.I.Lp,* specifies the set of all place nodes that are connected by an input link to a transition node, *t*. Note that the inner universal quantification is over each set defined by the outer universal quantification, *i.e*., for each transition node separately, it must be the case that all of its input place nodes have at least one token.

Execution predicates specify conditions under which a transition can occur and how the state is changed by the transition. The predicates in the table below form a complete description of the execution semantics of simple Petri Nets, but this need not be the case. The example in the next section is only part of the execution

Table 4: SPN Execution

Enable(pn:PetriNet) ==	$\forall t{:}pn.N.T\ [\ \forall p{:}t.I.Lp\ [\ p.NmTk \geq 1\ ] \Rightarrow t.Ste' = ENABLED\ ]$
StartFiring(pn:PetriNet) ==	$\exists\ t{:}pn.N.T$ $[\ t.Ste = ENABLED \Rightarrow \forall p{:}t.I.Lp\ [\ DeltaNmTks(p, -1)] \wedge$ $t.Ste' = FIRING]$
EndFiring(pn:PetriNet) ==	$\exists\ t{:}pn.N.T\ [\ t.Ste = FIRING \Rightarrow \forall p{:}t.O.Lp\ [DeltaNmTks(p,1)]$ $t.Ste' = DISABLED\ ]$
DeltaNmTks(p:P, i:INTEGER) ==	$[\ p.NmTk' = p.NmTk + i\ ]$

behavior will be specified.

2.0.3.3 Animation

The purpose of animation is to convey aspects of the behavior of the program to the programmer as it is executing.[1] The same predicate notation is now used to describe how visual attributes of the GBLE are coupled to state components and transitions involved in program execution. The following predicate specifies that all transition nodes should be highlighted when they are enabled.

$$\mathit{IllusEnabled(pn{:}PetriNet)} == \forall \mathit{t{:}pn.N.T}\ [\ \mathit{t.Ste} = \mathit{ENABLED} \Leftrightarrow \mathit{t.Gr.Hl} = \mathit{TRUE}]$$

This is a “simple” animation in which a component of the execution state of the program is reflected in a graphical component. We extend this basic concept of animation in the following way; instead of directly reflecting the state of the execution by the state of a graphical component, we also allow a *separate* animation program, identified by a function, to execute and change graphical components. Conceptually, the program runs, freezes at some point in its execution, an animation program runs and completes, and then the program continues. Thus the simple case above can be augmented.

$$\mathit{IllusEnabled2(pn{:}PetriNet)} == \forall \mathit{t{:}pn.N.T}\ [\ \mathit{t.Ste} = \mathit{ENABLED} \Rightarrow \mathit{t.Gr.Flash()}\ ]$$

Here *Flash* identifies an animation routine which repeatedly flashes the transition node (alternates the graphical component *HighLight* between true and false) until the transition node becomes disabled. The () indicates that *Flash* takes no parameters. We interpret the expression *t.Gr.Flash()* to be true if the routine has executed. Animation predicates can also be used to specify change of graphical attribute with *change* of state, for example, the predicate

$$\mathit{ShowTokenChange(pn{:}PetriNet)}\ == \forall \mathit{p{:}pn.N.P}\ [\ \mathit{p.NmTk} \neq \mathit{p.NmTk}' \Leftrightarrow \mathit{p.Gr.Hl} \neq \mathit{p.Gr.Hl}'\ ]$$

specifies that the highlighting of place nodes should change when a token is added or removed.

3.0 Example - Hierarchical Petri Nets

The previous section introduced GLIDE using simple Petri Nets as example. This section presents the GLIDE specification of a more complex variant of Petri Nets, PCM. PCM is a Petri Net based language intended for performance modelling of parallel computations[BA88] [Adi88]. It has a number of augmentations which help to illustrate more complex GLIDE specifications. A detailed understanding of the PCM model is not required. It is only being used as a real, more complex, example to illustrate features of GLIDE. The augmentations are the use of hierarchy and repeated instantiation via index parameters. Hierarchy is through the transition nodes, *i.e.*, a PCM transition node may be a composite node which contains a lower level subnet.

3.0.1 Productions

The following productions specify the structure PCM graphs:

PCMStructure	==	*PN*	*: PCMPetriNet;*
		SV	*: StateVariable**
PCMPetriNet	==	*P*	*: PlaceNo<de**;*
		T	*: TransNode**;*
		L	*: Link**;*
PlaceNode	==	*NmTk*	*: INTEGER;*
		I	*: Link**;*

1. The animation might also be after execution from a trace playback.

<table>
<tr><td></td><td></td><td>O</td><td>: Link**;</td><td></td></tr>
<tr><td></td><td></td><td>Gr</td><td>: Graphic;</td><td></td></tr>
<tr><td></td><td></td><td>Rp</td><td>: RepParameter</td><td>/* replication index parameter */</td></tr>
<tr><td>TransNode</td><td>==</td><td>Prt</td><td>: PrimitiveTransNode |</td><td></td></tr>
<tr><td></td><td></td><td>Sbt</td><td>: SubnetTransNode ;</td><td></td></tr>
<tr><td></td><td></td><td>I</td><td>: Link**;</td><td></td></tr>
<tr><td></td><td></td><td>O</td><td>: Link**;</td><td></td></tr>
<tr><td>PrimitiveTransNode</td><td>==</td><td></td><td>: TransAttibutes;</td><td></td></tr>
<tr><td></td><td></td><td>Gr</td><td>: Graphic</td><td></td></tr>
<tr><td>SubnetTransNode</td><td>==</td><td>Ta</td><td>: TransAttibutes;</td><td></td></tr>
<tr><td></td><td></td><td>It</td><td>: InitiatingTransition;</td><td></td></tr>
<tr><td></td><td></td><td>Tt</td><td>: TerminatingTransition;</td><td></td></tr>
<tr><td></td><td></td><td>PN</td><td>: PCMPetriNet;</td><td></td></tr>
<tr><td></td><td></td><td>Rp</td><td>: RepParameter</td><td>/* replication index parameter */</td></tr>
<tr><td></td><td></td><td>Gr</td><td>: Graphic</td><td></td></tr>
<tr><td>TransAttributes</td><td>==</td><td>Pi</td><td>: Pexpr;</td><td>/* predicate expression */</td></tr>
<tr><td></td><td></td><td>Phi</td><td>: Proc;</td><td>/* a procedure */</td></tr>
<tr><td></td><td></td><td>Tau</td><td>: Delay;</td><td>/* firing delay time*/</td></tr>
<tr><td></td><td></td><td>Ste</td><td>: State</td><td></td></tr>
<tr><td>State</td><td>==</td><td></td><td colspan="2">: ENABLED | FIRING | DISABLED / ACTIVE</td></tr>
<tr><td>RepParameter</td><td>==</td><td></td><td>: INTEGER</td><td></td></tr>
<tr><td>Link</td><td>==</td><td>Lt</td><td>: TransNode ;</td><td></td></tr>
<tr><td></td><td></td><td>Lp</td><td>: PlaceNode ;</td><td></td></tr>
<tr><td></td><td></td><td>Gr</td><td>: Graphic</td><td></td></tr>
</table>

There are two types of transition node *PrimitiveTransNode* and *SubnetTransNode*. Hierarchy is represented through the recursive use of the *PCMPetriNet* component in the *SubnetTransNode* production. The *RepParameter* attribute in *PlaceNode* and *SubnetTransNode* allows the programmer to specify how many times the node is replicated during execution.

The remaining productions specify the syntax of the expressions of components of transition nodes. They are predicates (*Pexpr*), arithmetic expressions (*Aexpr*), and procedures (*Proc*). Their syntax is taken directly from [BA88]:

<table>
<tr><td>Pexpr</td><td>==</td><td>: Pterm | Pexpr OR Pterm</td></tr>
<tr><td>Pterm</td><td>==</td><td>: Pfactor | Pterm AND Pfactor</td></tr>
<tr><td>Pfactor</td><td>==</td><td>: (Pexpr) | Patom | TRUE | FALSE | NOT Pfactor</td></tr>
<tr><td>Patom</td><td>==</td><td>: Aexpr = Aexpr | Aexpr ≠ Aexpr | Aexpr ≤ Aexpr |</td></tr>
<tr><td></td><td></td><td>Aexpr ≥ Aexpr | Aexpr > Aexpr | Aexpr < Aexpr</td></tr>
<tr><td>Aexpr</td><td>==</td><td>: Aterm | Aexpr + Aterm | Aexpr - Aterm</td></tr>
<tr><td>Aterm</td><td>==</td><td>: AFactor | Aterm MOD Afactor |</td></tr>
<tr><td></td><td></td><td>Aterm "/" Afactor | Aterm * Afactor</td></tr>
<tr><td>Afactor</td><td>==</td><td>: (Aexpr) | VAR | NUM | + NUM | - NUM</td></tr>
<tr><td>Proc</td><td>==</td><td>: Assnmt | Proc ";" Assmt</td></tr>
<tr><td>Assmt</td><td>==</td><td>: VAR "←" Aexpr</td></tr>
</table>

These productions make use of a third way of combining components; a space between components indicates combination by sequential concatenation, rather than choice or aggregation (as in a normal syntax notation).

3.0.2 Views

The basic view queries are similar to the ones used for SPNs. The table shows some specific to PCM.

Table 5: PCM Views

ShowLevelOneSubnets(pn:PCMPetriNet)==	$\{ x \mid x \in pn.T.Sbt.PN.P \vee x \in pn.T.Sbt.PN.T \vee$ $x \in pn.T.Sbt.PN.L \}$
ShowNumTokens(pn:PCMPetriNet)==	$\{ x, x.NmTk \mid x \in pn.P \}$
ShowPredicates(pn:PCMPetriNet) ==	$\{ x, x.Ta.Pi \mid x \in pn.T \}$
ShowProcedures(pn:PCMPetriNet) ==	$\{ x, x.Ta.Phi \mid x \in pn.T \}$
ShowDelays(pn:PCMPetriNet) ==	$\{ x, x.Ta.Tau \mid x \in pn.T \}$ **changed**
ShowActiveSubNets(pn:PCMPetriNet) ==	$\{x \mid x \in pn.T@.Sbt.PN.T \wedge x.Sbt.Ta.Ste = ACTIVE \}$

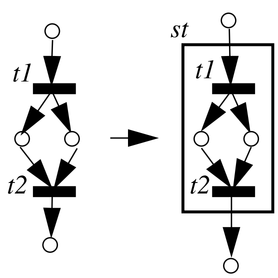
The first query *ShowLeveOnelSubnets* illustrates how to specify a view which displays the nodes and links of all the subgraphs one level down in the hierarchy. The "dot-addressing" notation path expression selects the subgraphs of the transition nodes which are of type *SubnetTransNode*. The middle four queries show the whole graph (at one level) with all nodes labeled with the value of some common attribute of the nodes or links (associated number of tokens (*NmTk*), predicates (*Pi*), delays (*Tau*), or procedures (*Phi*)). A graph display that can quickly switch between each attribute is very useful. It is found, for example, in[Hun90] where nodes also have a large number of attributes. In the current specification language, a separate query specification is needed for each attribute.

The last view, *ShowActiveSubNets*, states that the set of all subnet transition nodes that are in the state *ACTIVE*, at *any* level in the hierarchy, should be displayed. This is achieved by specifying a recursion in the query. Recursion is specified by the @ symbol. In this query it indicates that the set of elements x is generated by taking the initial set of top level transition nodes ($x \in pn.T$) and then generating a transitive closure with the "function" *.Sbt.PN.T* . Of all those elements only those for which $x.Sbt.Ta.Ste = ACTIVE$ is true are members of the view.

3.0.3 Editing

The basic editing operations of PCMs are similar to those shown for SPNs. We show here a more complex editing operation which allows the user to aggregate nodes at one level and put them in a subnet at a lower level. Such an operation would normally be performed via a "marquee" - encircling the nodes to be aggregated with a mouse cursor.

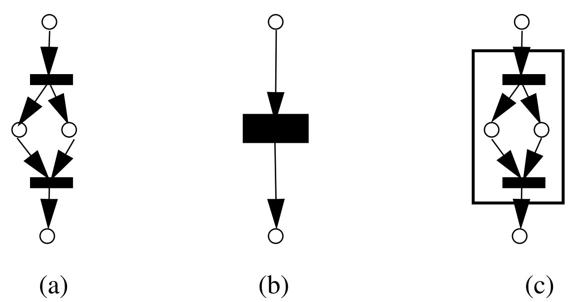
Table 6: PCM Editing

	*Aggregate (pn:PCMPetriNet,* *p:PlaceNode**,* *t:TransNode**,* *l:Link**,* *t1:TransNode,* *t2:TransNode) ==*	$[\ st = new(SubnetTransNode) \wedge$ $st.It' = t1 \wedge$ $st.Tt' = t2 \wedge$ $st.PN.P' = p \wedge$ $st.PN.T' = t \wedge$ $st.PN.L' = l \wedge$ $pn.P' \cup \{p\} = pn.P \wedge$ $pn.T' \cup \{t\} = pn.T \wedge$ $pn.L' \cup \{l\} = pn.L\ ]$

Aggregate replaces a set of nodes and links with a new *SubnetTransNode, st*, and changes the set of nodes to be a subgraph of *st*. The arguments to *Aggregate* are: the whole original graph, *pn*, the set of place nodes encircled, *p*, the set of transition nodes encircled, *t*, the set of links encircled, *l,* the transition node that becomes the initiating node of the subnet, *t1*, and the transition node that becomes the terminating node of the subnet, *t2.* The notation used in the parameter specification, e.g., *(p:PlaceNode**)*, indicates that *p* is a set of elements of type *PlaceNode* rather than just a single element of type *PlaceNode*. The last three equalities of the predicate also specify that the selected elements placed in the subgraph are removed from the original graph. This operation is analogous to the hierarchical structuring operation for the CODE system specified algebraically in [ea91]. The advantage of an algebraic description of such operations is that it facilitates the development of algebraic expressions to describe complete programs, based on their composition.

The distinction between view queries and editing operations is important and can be quite subtle as Figure 3 illustrates. The aggregation of a set of nodes into a single composite node is an editing operation available to the user (a → b or a → c). Despite it being an editing operation, the invocation of aggregation does not change the semantics of the program, since the execution of the program with or without the aggregation is identical (in PCM). Furthermore, the aggregated program can be displayed "transparently"[1](c). This is a change of view specification, not editing, since no part of the program is altered (b ↔ c). The (c) display can be characterized as graphically displaying *both* a non-terminal component (*SubnetTransNode*) *and* the components of one of its components (the nodes and links of *SubnetTransNode).*

FIGURE 3. Edit Change vs. View Change in Hierarchical Petri Net Model



1. The term "white box" is used in [PT90] to differentiate it from an opaque "black box" view.

3.0.4 Execution

The following predicates describe the basic execution semantics of PCM. The predicates were derived from

Table 7: PCM Basic Execution

Enablement(pn:PCMPetriNet) ==	$\forall t{:}pn.T,\ \forall p{:}t.I.Lp$ $[\ p.NmTk > 0 \wedge t.Pi(pn.SV) = TRUE \Rightarrow t.Ta.Ste' = ENABLED]$
Disablement(pn:PCMPetriNet)==	$\forall t{:}pn.T,\ \forall p{:}t.I.Lp$ $[\ p.NmTk > 0 \wedge t.Pi(pn.SV) = FALSE \Rightarrow t.Ta.Ste' = DISABLED]$
PrtFiring(pn:PCMPetriNet) ==	$\exists t{:}pn.T.Prt$ $[\ t.Ta.Ste = ENABLED \Rightarrow t.Ta.Ste' = FIRING \wedge$ $\forall p{:}t.I.Lp\ [DeltaNmTk(p,-1)]\ ]$
SbtFiring (pn:PCMPetriNet)==	$\exists t{:}pn.T.Sbt$ $[\ t.Ta.Ste = ENABLED \Rightarrow t.Ta.Ste' = ACTIVE \wedge$ $t.It.Ta.Ste' = FIRING \wedge$ $\forall p{:}t.I.Lp\ [DeltaNmTk(p,-1)]\ ]$
Delay (pn:PCMPetriNet)==	$\exists t{:}pn.T$ $[\ t.Ta.Ste = FIRING \wedge t.Ta.Tau(pn.SV) \Rightarrow t.Ste' = DISABLED \wedge$ $\forall p{:}t.O.Lp\ [DeltaNmTk(p,1)]$ $\wedge\ t.Ta.Phi(SV)\ ]$

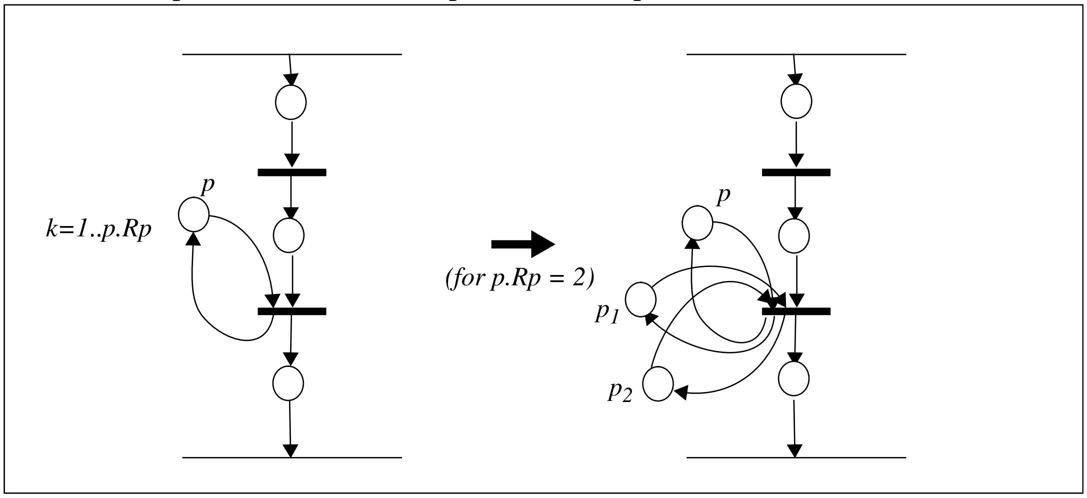
the description of execution semantics for primitive transition nodes and subnet transition nodes as described in [Adi88]. They are similar to the ones for the SPN model but they include several extensions. Transition nodes in PCM have the requirement that the predicates associated with them (*Pi*) should be true for the nodes to become enabled. They must also execute a procedure (*Phi*) and there is a delay (*Tau*) before tokens “arrive” at place nodes. We specify these extensions by including them in the predicates and interpreting them as having the boolean value true when their execution completes (C.f. animation routines). There is also a separate predicate for subnet transition node execution, *StFiring*. It prescribes that the initiating node (*st.It.Ta.Ste*) should be set to firing when its subnet is active.

This examples illustrates limiting the amount of detail in which the execution is described. In particular, the execution of procedures and evaluation of predicates are viewed as atomic. If more detailed animations were required, such as displaying the values of subexpressions in the procedures or predicates, then more execution predicates would be required to express this detail.

3.0.4.1 Replication

A further complication in PCM is that nodes and subnets may be created dynamically during execution by replication. The diagram in Figure 4 illustrates a simple example in which a place node is replicated.

FIGURE 4. Expansion of a subnet and place node with parameters (from [PCM878b])



The predicate *PlaceExpansion* shows how such dynamic graph behavior can be captured in a transition

Table 8: PCM Expansion Execution

PlaceExpansion(p:PlaceNode) ==	$\forall i{:}1..\#p.I,\ \forall j{:}1..\#p.O,\ \forall k{:}1..p.Rp$ $[\ p_k = new\ (PlaceNode) \wedge l_{ik} = new(Link) \wedge l_{jk} = new(Link) \wedge$ $l_{ik} \in p_k.I' \wedge l_{jk} \in p_k.O' \wedge$ $\forall t{:}p.I.Lt\ [\ l_{ik} \in t.O'\ ] \wedge \forall t{:}p.O.Lt\ [\ l_{jk} \in t.I'\ ]\ ]$

predicate. The predicate uses subscripting to specify how this replication occurs. It specifies that a place node is replicated a number of times specified by its replication parameter value (*p.Rp*), and all incoming and outgoing links are also replicated. The ellipsis marks ("..") in a restricted quantification is to indicate the introduction of an *index* variable over the range of a component, hence the expression $\forall k{:}1..p.Rp$, indicates that *k* takes on all integer values from 1 to *p.Rp*, and the expression $\forall i{:}1..\#p.I$, takes on the values from 1 to the number of elements ("#") in the set *p.I, i.e.*, the number if input links that place node *p* has.

3.0.5 Animation

Table 9: PCM Animation

HighlightActiveSubNets(pn:PCMPetriNet) ==	$\forall t{:}pn.T@.St.PN.T$ $[\ t.St.Ta.State = ACTIVE \Rightarrow t.St.Gr.Hl = TRUE]$

The animation specification of the *HighlightActiveSubNets* animation predicate shows how to specify an animation which highlights transition nodes at any level in the graph which are in the state *ACTIVE*. The predicate is almost identical to the last view specified in Table 5. It uses the @ construct to indicate the same recursion.

4.0 Compilation Sketch

This section describes how GLIDE specifications are compiled. The main task of the compilation algorithm is to translate the abstract description of the graph as given in the grammar into concrete data types and to translate the declarative description of actions as given in the predicates into a procedural form. The target of compilation is some GUI library. Since there is still quite a large variation in the level of functionality provided by various GUI packages, we anticipate having a retargetable back-end which can generate code for different ones. We illustrate the compilation process here with examples emitting code of the EDGE graph editing library programming interface [PT90]. The main interface functions provided by EDGE are described in the Section 1.1. The compilation of the simple Petri Net specification is illustrated in Sections 4.2-4.3 and summarized in Section 4.4.

The final appearance and interaction of a system will depend, to a large extent, on the specific GUI package used. All GUI libraries systems may be restricted in some way - from not having "forms" for displaying and editing node attributes to not having any support for graph display manipulation at all. Libraries may or may not provide "widgets" such as menus, tables, or text editing windows. If the library does provide graph display and manipulation, it may not be adequate. For example, one important feature of several GBLs is the use of nodes with "ports". Ports are used to provide local names for separate links into or out of a node. In a GUI which does not support complex nodes with ports that can be selected and labeled, a substitute means would be used such as listing the ports in a pop-up form with the other attributes of a node.

The compilation process is separated following the decomposition of the specification language described in Section 1.0, *i.e.*, productions, views, and predicates, and within predicates following editing, execution and animation. In order to compile the language, some (strong) restrictions on the form of the predicate and query expressions are currently being made. The Petri Net and PCM programming environment specifications fall within these restrictions, but we anticipate relaxing some of the restrictions for future more complex applications. The restrictions are noted in each section.

4.1 Target Language

EDGE is a graph manipulation and display package written in C++. It provides a set of functions for manipulating and displaying graph data structures that is typical of graph GUIs. There are functions for:

1. Instantiation of objects from `graph`, `node` and `edge` classes
```
n = new node;
```
2. Component access - lists of nodes or edges can be retrieved by functions of the form
```
graph->getheadnode(); /* get head of list of all nodes */
node->getnextnode();
graph->getheadedge();
edge->getnextedge();
node->getsucclist();   /* get children of a node */
```
3. Attribute access - values of attributes can be retrieved by functions of the form:
```
edge->getlabel();
edge->setlabel("mynode");
```

EDGE nodes have a fixed set of graphical attributes (`icon`, `label`, `font`, `border`, `color`, `size`, etc.). Application attributes are included by specializing the `node` or `edge` class, and can be accessed by providing access functions similar to (3).

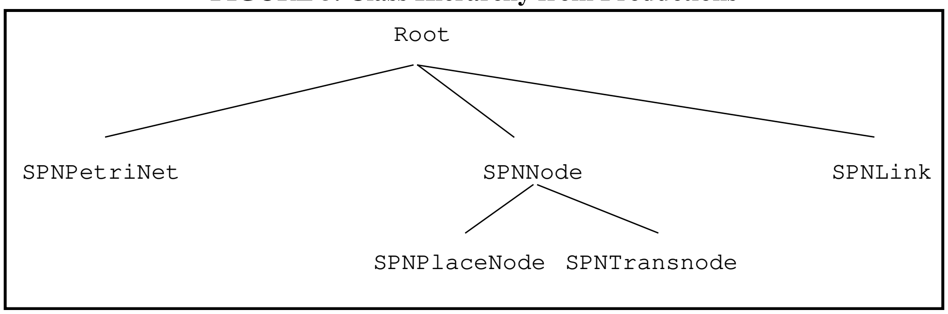
4.2 Productions

The first part of the compilation is to parse the productions. The components of GLIDE productions are translated into classes of an object-oriented language. The productions are parsed to generate the required classes and attributes. A *choice* ("|") in a production is mapped to a *class-subclass* relationship and an *aggregation* (“;”) in a production is mapped to an *instance variable* of a class. Taking the Standard Petri Net as example, the productions:

PetriNet	==	*N*	*: PNNode**;*
		L	*: PNLink***
PNNode	==	*P*	*: PlaceNode* |
		T	*: TransNode*

are parsed to generate a class for each component:

FIGURE 5. Class Hierarchy from Productions



Class `SPNNode` is the superclass of the classes `SPNPlaceNode` and `SPNTransNode`. The first production can be represented by a class `SPNPetriNet` having two instance variables, one of type `SPNNode` containing the list (or some other concrete implementation of a set) of nodes and one of type `SPNLink` containing the list of links.

The classes may be generated as subclasses of the classes provided by an object-oriented GUI such as EDGE, integrated via multiple inheritance, or created separately with routines to maintain consistency between state in the GUI instances and the instances of the separate classes. For this description of compilation we assume that the classes shown above are created as subclasses the of `graph`, `node` and `edge` classes provided by EDGE. The classes and instance variables generated by this process are entered in a symbol table which is then used to resolve the references to these components (dot-addresses) in the predicates and views.

4.3 Views

A view specification is translated into a sequence of instructions to retrieve instances and instance variable values created under the class hierarchy. The restriction on view expressions is that they consist of a disjunction of membership expressions. For example, the query specifying the top-level view for Standard Petri Nets:

$$WholeGraph(pn{:}PN) == \{x \mid x \in pn.N.P \vee x \in pn.N.T \vee x \in pn.L\}$$

can be translated directly into a sequence of three loops which display the place, transition, and link components(EDGE derived methods are italicized in the example translations of the next sections.):

```
wholegraph(SPNPetriNet* pn)
{
  SPNPlaceNode* p;
  p = pn->getheadnode();
  while (p != NULL ) {
      if (p->typename == "PlaceNode") { p->display();}
      p = p->getnextnode();
  }
```

```
 SPNTransNode* t;
 t = pn->getheadnode()
 while (t != NULL ) {
     if (t->typename == "TransNode") { t->display();}
     t = t->getnextnode();
 }

 SPNLink* l;
 l = pn->getheadedge()
 while (l != NULL ) {
     l->display();
     l = l->getnextedge();
 }
}
```

4.4 Predicates

Predicates are compiled by generating a control flow, based on the parse of the logic formulas, that is consistent with the data dependencies implicit in the formulas. The data dependencies are deduced from the type of predicate (editing, execution, or animation), the priming annotations, and the logical operators in the formula.

4.4.1 Editing

The editing predicates specify user-initiated operations. The operations would typically be made available in a command menu and their arguments identified by mouse selection or provided by type-in. The restriction made on the syntax of editing predicates is that they consist of a conjunction of equality expressions. Each equality is translated into an assignment. Thus the compilation takes the declarative statement with four conjuncts:

$$AddTrans(p{:}PlaceNode) == l = new(PNLink) \wedge t = new(TransNode) \wedge p.O' = p.O \cup \{l\} \wedge t.I' = \{l\}$$

and maps into procedural code with four corresponding sets of statements:

```
AddTransNode(SPNPlaceNode* p)
{
 SPNLink* l;
 SPNTransition* t;

 l = new SPNLink;
 l->settypename("Link");

 t = new SPNTransNode;
 t->settypename("TransNode");

 p->addsucc(t);
 l->setsourcename(p->gettitle());

 t->addprec(p);
 l->settargetname(t->gettitle());
}
```

The conjuncts of the predicate may be in any order. The order of execution of corresponding statements, however, must not violate any data dependencies. This is ensured by sorting conjuncts such that instantiations (`new`) are first and primed equalities are last. Set operators such as union ("∪") are mapped to the operations appropriate for the corresponding concrete implementation, such as list concatenation.

4.4.2 Execution

Execution predicates specify the changes that occur to a program when it is in a given state. The execution predicates are compiled into a condition→action form; the condition part consists of all the atomic tests on the state and the action part consists of all the specified changes to the state. Thus, execution predicates are restricted to be in the form of an implication (“⇒”) or an iff (“⇔”). The left side of an implication may not contain any primed variables and the rhs contains only variables bound to newly instantiated components or primed variables. The rules for the execution semantics of Petri Nets follow these restrictions. Thus:

$$Enable(pn{:}PetriNet) == \forall t{:}pn.N.T\ [\ \forall p{:}t.I.Lp\ [\ p.NmTk \geq 1] \Rightarrow t.Ste' = ENABLED\ ]$$

is translated to:

```
enablement(SPNPetriNet* pn)
{
  SPNTransNode* t;
  SPNPlaceNode* p;
  bool EnabledFlg;

  t = pn->getheadnode();
  while ( t != NULL ) {                    /* from ∀t */
            if (t->typename == "TransNode") {
                  EnabledFlg = TRUE;
                  p = t->getheadsuccnode();
                  while (p != NULL ) {     /* from ∀p */
                        EnabledFlg = EnabledFlg && p->getNumTokens(y) >= 1);
                        p = t->getnextsuccnode();
                  }
                  if (EnabledFlg == TRUE) {
                        t->setState(ENABLED);
                  }
            }
  t = t->getnextnode();
  }
}
```

The nested universal quantifications translate directly into the nested while loops. Some procedures do not invoke others and are thus only connected by their writing and reading of the shared state variables (such as *t.Ste*). Firing is defined by three predicates. The two *StartFiring* and *EndFiring* both invoke the predicate *DeltaNmTks*:

$$StartFiring(pn{:}PetriNet) ==$$
$$\exists\ t{:}pn.N.T\ [\ t.Ste = ENABLED \Rightarrow \forall p{:}t.I.Lp\ [\ DeltaNmTks(p, -1)] \wedge t.Ste' = FIRING]$$

is translated to:

```
startfiring(SPNPetriNet* pn)
{
  SPNTransitionNode* t;
  SPNPlaceNode* p;

  until (t->getState == ENABLED){
      t = RandomSelectFromList(pn->getTransitionNodes)
  }
  p = t->getheadprednode();
  while (p != NULL ) { /* from ∀p */
      DeltaNmTks(p,-1);
      p = t->getnextprednode();
  }
  t->setState(FIRING);
}
```

Here, the function `RandomSelectFromList` is used to implement the non-deterministic choice (*t:pn.N.T*) of transition node to fire. The predicate *EndFiring*:

$$EndFiring(pn{:}PetriNet) == \exists\, t{:}pn.N.T\ [\ t.Ste = FIRING \Rightarrow \forall p{:}t.O.Lp\ [DeltaNmTks(p,1)] \wedge t.Ste' = DISABLED\ ]$$

translates to a similar pair of loops:

```
endfiring(SPNPetriNet* pn)
{
  SPNTransitionNode* t;
  SPNPlaceNode* p;

  t = pn->getheadnode();
  while ( t != NULL || t->getState != ENABLED) {/* from ∀t */
      t = t->getnextnode();
  }
  if (t->getState == ENABLED) {
      p = t->getheadsuccnode();
      while (p != NULL ) {                /* from ∀p */
            DeltaNmTks(p,1);
      p = t->getnextsuccnode();
  }
  t->setState(DISABLED);
}
```

This last predicate *DeltaNmTks*:

$$DeltaNmTks(p{:}P,\ i{:}INTEGER) == [\ p.NmTk' = p.NmTk + i\ ]$$

translates to a pair of statements to access and update the attribute recording the number of tokens:

```
DeltaNmTks(PlaceNode* p, int i)
{
  int NumTk;

  NumTk = p1->getNumTokens();
  p1->setNumTokens(NmTk + i);
}
```

4.4.3 Animation

The animation predicates are translated into functions calls which change specified graphical attributes when the execution reaches a state prescribed by the predicate. Hence, the predicate:

$$IllusEnabled(pn{:}PetriNet) == \forall t{:}pn.N.T\ [\ t.Ste = ENABLED \Leftrightarrow t.Gr.Hl = TRUE]$$

is translated to:

```
illusenabled(SPNPetriNet* pn)
{
TransNode* t;
t = pn->getheadnode(); /* the graph, pn, available globally */
while ( t != NULL ) {                    /* from ∀t */
  if (t->typename == "TransNode") {
      if (t->getState() == ENABLED ) {
            t->Highlight();
      } else {                           /* from ⇔ */
            t->UnHighlight();
  }
  t = t->getnextnode();
} }
```

In order to include animation in the execution, the invocations of the animation procedures must alternate with the invocation of the execution procedures, so that state change can be detected and displayed. This is inefficient and can improved by an optimization phase so that calls to the highlight function are made directly when the variable is modified by an execution procedure.

4.5 Compilation Algorithm

This section summarizes the algorithm illustrated by the previous examples. The specification consists of a set of *productions* (P), a set of *queries* (Q), and a set of *predicates* calculus formulas (F). Each production contains a set of tagged *components.* Queries consist of a set of dummy variables and a disjunctive logic formula. Each disjunct is a membership expression. A predicate consists of either an implication or a conjunctive formula. If the predicate is an implication, its rhs must be a conjunctive formula. These conjuncts must be equality expressions.

Algorithm:

1. [Initialize for Productions] Create the Production Symbol Table (T_{pro}) to hold the names of the components in the productions. T_{pro} includes: the tag name, the component name, the name of its corresponding class, its parent or the class in which it is an instance variable, and the instance variables it contains.
2. [Parse Productions] For each production p_i in P, create a class for the production name (the lhs) and make an entry into T_{pro} with the class name and the production name.

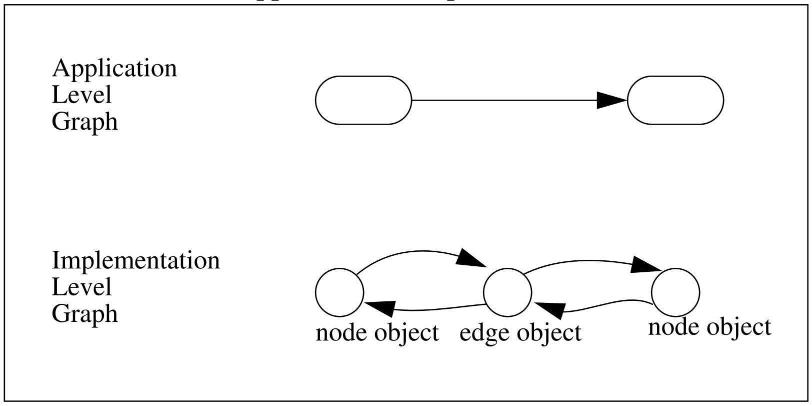
   For each component (rhs) of p_i, create a class (unless, checking in T_{pro}, one has previously been created). If the components are combined by choice (“ | ”), create a subclass of the production lhs class for each. If the components are combined by aggregation (“ *;* ”), add an instance variable to the production lhs class. Add and entry for each component into T_{pro} with the name of its tag and the name of its class or instance variable.

   This completes the parsing of the productions.
3. [Initialize for Queries] Create the Queries Symbol Table (T_q) which lists the queries. T_q contains the name of the query and the list L_c of components to be retrieved for that query.
4. [Parse Queries] Parse each query and add to L_c a pointer into T_{pro} which identifies the components specified by the membership expressions of the query.
5. [Queries Code Generation] For each component in $L_{c,}$ generate a loop which retrieves the instances or instance variable contents with the access functions provided by the target language.

   This completes compiling the queries.
6. [Initialize for Predicates] Create the Predicates Symbol Table (T_{pre}) table which lists the predicates and variables. A predicate entry in T_{pre} includes the predicate name, the name of its corresponding procedure, a pointer to a table of local variables, the input variables, and the output variables. A variable entry in T_{pre} indicates whether the variable is input or output and has a pointer to the variables class or instance variable in T_{pro}.
7. [Parse Predicates] For each predicate f_i in F, create the symbol table T_{fi} for listing the local variables of the predicate, with a pointer to their class in T_{pro}. Parse each predicate and generate its parse tree. Enter into T_{pre} the arguments of the predicate as input variables and the primed variables as output variables, and enter into T_{fi} the remaining variables as local variables.
8. [Ordering] Reorder the conjuncts in the parse tree so that no data dependencies are violated The data dependencies can be deduced from the variable properties listed in the symbol tables T_{pre} and T_{fi}.
9. [Predicates Code Generation] Traverse the re-ordered parse tree of each predicate. If the predicate is an implication, emit for the lhs the corresponding boolean test expression which invokes the target language access functions. For each equality conjunct in the conjunctive formula of the predicate (*i.e.*, in the rhs if the predicate is an implication), emit the sequence of target language assignments that it expands to.

This completes the compilation. Note that there are two separate graph structures: the complex low one level implementation data structure that is generated and used by the target library, and the higher level abstract graph described by the specification. For example, the EDGE library maintains links from each sibling node to the next sibling node. The graph at the higher level of abstraction is supported by a graph datastructure at the lower level is more complex, just as the declarative predicates are expanded into a larger set of procedural statements. The following diagram illustrates this.

FIGURE 6. Application vs. Implementation Structure



5.0 Conclusions

This paper has presented a high level specification language for describing graph based programming environments and illustrated its use with two simple examples. The language consists of productions to specify the syntax of the language and its environment, set specifications to describe views of programs written in the language, and state transition predicates to describe the semantics of editing, execution and animation. The basic steps involved in compiling these specifications have been described. We believe that this formalization of GBLEs provides a sound basis for our goal of automating the generation of these systems. Though we have used these simple examples our language is geared to the more complex systems.

This work has focussed on developing the specification language. We have used it to describe six other existing systems and based on this we believe that it provides sufficient expressive power to cover a wide range of GBLEs. In contrast to other formalizations of the syntax and semantics of visual programming languages our abstractions do not incorporate positional information. This is because (as noted in[Szw87]), in most graph interfaces, its is the interconnection of nodes and links that conveys information, but their locations are immaterial. There are applications of graph interfaces where location alignment do carry meaning, but in most GBLs this is not the case. The use of graphs in which location and alignment are relevant has been studied by [Mar90].

We have intentionally left many issues of appearance of graphical display open. This level of abstraction allows us capture GBLEs in a way that is independent of a particular GUI implementation. This is also a way of not precluding displaying the graph in novel ways, e.g. using 3D graphics, or indeed in a completely textual form. This work follows recent trends in user interfaces generation from domain models, as exmplified by the ITS of Wiecha et al.{}, ACE of Johnson{} and foo of Foley in al {}. These systems place great emphasis on generating interfaces from the semantics of the application and domain data model, in our case we are providing a means for creating data models in our targeted domain, programming.

There are existing systems which are similar to ours in that they provide a general graph based programming language which can be refined to specific applications such as the PFG system of [Sto88] and the Olympus system of [Nut91]. Both of these systems are however tied to an underlying Petri Net model of computation; we are attempting to be more general. The “h-graph selectors” used in PFG is similar the dot-addressing scheme we use, though selectors describe a path through a hierarchical graph instance rather than its schema. Work by [Got89] describes graph interfaces that use and are specified by graph grammars.

An important concept developed in this paper is the use of productions to describe both the graph structures and associated attributes in a uniform way. This approach is appropriate because many GBLs “bottom-out” in text, *i.e.*, the attributes of the nodes are expressions which can be described with the same productions (*e.g.*, the syntax of node firing rules). This abstraction allows the specifier to easily migrate the display and manipulation of the program between the graphical and textual forms. The uniform way in which semantics is expressed, be it for editing, execution, or animation, is also important. It, allows, for example, the development of systems in which all three activities can occur concurrently (e.g., execution of partially written programs).

6.0 Acknowledgments

Thanks to JCB, RR, TC, BdC($$), PN, HFK($$), IDB (bib), DB, JW, RB, AC - Practical Needs

7.0 References

[Adi88] Ashok Adiga. *Performance Modelling of Parallel Computations*. PhD thesis, Dept. of Comp. Sci., Univ. of Texas at Austin, 1988.

[BA88] J.C. Browne and Ashok Adiga. *Performance Evaluation of Supercomputers*, chapter -Graph Structured Performance Models-. Elsevier, 1988.

[Bar86] P. S. Barth. An object-oriented approach to graphical interfaces. *ACM Transactions on Graphics Special Issue on User Interface Software*, 5(2):142–172, April 1986.

[BCL90] Duane Bailey, Janice Cuny, and Craig Loomis. Paragraph: Graph editor support for parallel programming environments. *International Journal of Parallel Programming*, 19(2):75–110, 1990.

[BDG+91] A. Beguelin, J. Dongarra, G. Geist, R. Manchek, and V. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Supercomputing '91*, pages 435–444, Nov 1991.

[BN91] A. Beguelin and G. Nutt. Collected papers on Phred. Tech Report CU-CS-511-91, Dept. Comp. Sci. Univ. Colorado at Boulder, January 1991.

[Bro85] J. C. Browne. Formulation and programming of parallel computers: A unified approach. In *Proc. Intl. Conf. Par. Proc.*, pages 624–631, 1985.

[Bro88] Marc H. Brown. *Algorithm Animation*. ACM Doctoral Dissertation Award. MIT Press, Cambridge, Mass, 1988.

[Che91] D. Cheng. A survey of parallel programming tools. Technical Report RND-91-005, NASA Ames, May 1991.

[CL88] C. Chow and S. Lam. Prospec: An interactive programming environment for designing and verifying communication protocols. *IEEE Transactions on Software Engineering*, 14(3):327–338, March 1988.

[CS90] L. Chang and B. Smith. Classification and evaluation of parallel programming tools. Tech Rept CS90-22, Dept. of Comp. Sci. Univ. of New Mexico, 1990.

[Dye89] R. Dye. Labview : A visual data-flow programming language and environment. Master's thesis, Dept. of Elec. and Comp. Eng. University of Texas at Austin, 1989.

[ea85] D. Notkin et al. The Gandalf project. *The Journal of Systems and Software*, 1985.

[ea91] J. Werth et al. The interaction of the formal and practical in parallel programming environment development: Code. Tech. Rep. TR-91-09, Dept. Comp. Sci., Univ. Texas at Austin, 1991.

[EGF91] M. Blattner E. Glinert and C. Frerking. Visual tools and languages: Directions for the 90's. In *Workshop on Visual Languages*, pages 89–95. IEEE, 1991.

[FA90] D. Workman F. Arefi, C. Hughes. Automatically generating visual syntax-directed editors. *Communications of the ACM*, 33(3), March 1990.

[FJ86] F. Feldbrugge and K. Jensen. Petri Net tools survey. Petri Nets: Central Models and Properties. In *Advances in Petri Nets*, pages 20–61, Berlin, 1986. Springer-Verlag. LNCS 254.

[GNV88] E. Gansner, S. North, and K. Vo. Dag - a program that draws directed graphs. *Software Experience and Practice*, 18(11):1047–10621047–1062, November 1988.

[Got89] H. Gottler. Graph grammars, a new paradigm for implementing visual languages. In *Eurographics'89*, pages 505–516, 1989.

[Har86] D. Harel. On Visual Formalisms. *Communications of ACM*, 31(5):514–529, May 1988.

[Hil92] D.D. Hills. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing*, 3:69–101, June? 1992.

[Hun90] N. Hunt. Idf: A graphical data flow language for image processing and computer vision. In *IEEE Conf. on Systems, Man, and Cybernetics*. IEEE, Nov 1990.

[Jac85] R. J. K. Jacob. A state transition diagram language for visual programmingformulation and programming of parallel computers: A unified approach. *Computer*, 18(8):51–59624–631, August 1985.

[JG89] David Jablonowski and Vincent Guarana. Gmb - a tool for manipulating and animating graph structures. *Software Practice and Experience*, 19(3):283–301, March 1989.

[JM88] A. Jahanian and A. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 1988.

[KG88] Michael F. Kleyn and Paul C. Gingrich. GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views. In *Proceedings of the Third ACM Conference on Object Oriented Systems, Languages, and Applications*, pages 191–205, September 1988.

[KS90] A. Karrer and W. Scacchi. Requirements for an extensible object-oriented tree/graph editor. In *Proceedings of ACM Third Annual Symposium on User Interface Software and Technology*, pages 84–91, 1990.

[Lam90] L. Lamport. A temporal logic of actions. Tech. Report 57, Digital Research Center, 1990.

[LER92] T.G. Lewis and H. El-Rewini. *Introduction to Parallel Computing*, chapter 12. Prentice-Hall, 1992.

[Mal82] William R. Mallgren. *Formal Specification of Interactive Graphics Programming Languages*. PhD thesis, Univ. of Washington, 1982. ACM Distinguished Dissertation.

[Mar90] J. Marks. A syntax and semantics for network diagrams. In *IEEE Workshop on Visual Languages*, pages 104–110, Skokie Illinois, 1990.

[Mey90] B. Meyer. *Introduction to the Theory of Programming Languages*. Prentice Hall, 1990?

[MHC89] J. Vlissides M. Hinton and P Calder. Composing user interfaces with interviews. *IEEE Compute*, 22(2):8–22, February 1989.

[MK92] J. Magee and J. Kramer. Mp: A programming environment for multicomputers. In *Proc. of Working Conference on Programming Environments for Parallel Computing*, Edinburgh, April 1992. Springer-Verlag. IFIP WG10.3.

[NB92] P. Newton and J. C. Browne. The code2.0 graphical parallel programming language. In *Proc. ACM Intl. Conf on Supercomputing*, July 1992.

[Nut91] G. Nutt. A simulation sytem architecture for graph models. In *Advances in Petri Nets*, Berlin, 1991. Springer-Verlag. LNCS ???

[Pau90] Frances Newberry Paulisch. *The Design of an Extendible Graph Editor*. PhD thesis, Institute for Informatics, University of Karlsruhe, March 1990.

[Pet81] J. Peterson. *Petri net theory and the modeling of systems*. Prentice-Hall, 1981.

[PT90] Frances Newberry Paulisch and Walter F. Tichy. Edge: An extendible graph editor. *Software-Practice and Experience*, S1(20), June 1990.

[RDM+87] L. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, and A. Tuan. A browser for directed graphs. *Software Practice and Experience*, 17(1):61–76, 1987.

[Rep84] T. Reps. *Generating Language-Based Environments*. ACM Doctoral Dissertation Award 1983. MIT Press, Cambridge, Mass, 1984.

[Sil92] Silicon Graphics Inc. *Iris Explorer‘ User’s Manual*, Jan 1992.

[Spi88] J. Spivey. *Understanding Z - A Specification Language and its Formal Semantics*. Cambridge Univ. Press, 1988.

[SSW+92] D. Szafron, J. Schaeffer, P.S. Wong, E. Chan, P. Lu, and C. Smith. The enterprise distributed programming model. In *Proc. of Working Conference on Programming Environments for Parallel Computing*, Edinburgh, April 1992. Springer-Verlag. IFIP WG10.3.

[Sta90] John T. Stasko. The path-transition paradigm: A practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1:213–236, 1990.

[Sta91] Stardent Computer Inc. *AVS Reference Manual*, 1991.

[Sto88] D.P. Stotts. The PFG Language: Visual Programming for Concurrent Computation Expressing High-Level Visual Concurrency Structures in the PFG Kernel Language. In *Int. Conf. on Par. Proc.*, pages 72–79, August 1988. Vol2: Software, Univ Maryland.

[Szw87] Gerd Szwillus. Cegs - a system for generating graphical editors. In H. Bullinger and B. Schakel, editors, *Human-Computer Interaction - Interact '87*. Elsevier, 1987.

[Tho90] Gregory S. Thomas. Xsim 2.0 user's guide. ftp cs.washington.edu, April 1990. Dept. of Comp Sci. Univ Washington.

[TR81] T. Teitelbaum and T. Reps. The cornell program synthesizer. *Communications of the ACM*, 24(9):563–573, Sept. 1981.

[VL89] John Vlissides and Mark Linton. Unidraw: A framework for building domain-specific graphical editors. In *SIGGRAPH User Interface Symposium '89*. SIGGRAPH, 1989.

8.0 Appendix

This appendix describes the syntax of the GLIDE, the language used to specify GBLEs. We use standard BNF notation, plus [ ] to indicate optional components and | to indicate alternatives. There are three sections, one for each distinct part of the language, but common productions already defined in a previous section are not repeated. For clarity we have omitted the additional subscripting index notation for dynamic graphs (as used in Table 8).

8.1 Production Syntax

```
<production>          ::= identifier == <productionrhs>
<productionrhs>       ::= [<tag>] ‘:’ identifier <superscript> [<connector> <productionrhs>]
<tag>                 ::= identifier
<connector>           ::= ‘|’ | ‘;’ | <space>
<superscript>         ::= ‘*’ | ‘**’ | ‘***’
<space>               ::= ‘ ’
```

8.2 View Query Syntax

```
<query>               ::= <lhs> ‘==’ ‘{’ <cvarlist> ‘|’ <qlformula> ‘}’
<lhs>                 ::= identifier [ ‘(’ <paramlist> ‘)’ ]
<paramlist>           ::= identifier ‘:’ <dotaddress> [ ‘,’ <paramlist> ]
<cvarlist>            ::= var [ ‘,’ <cvarlist> ]
                      |   var ‘.’ <dotaddress> [ ‘,’ <cvarlist> ]
<dotaddress>          ::= identifier [ ‘.’ <dotaddress> ]
<qlformula>           ::= <membership> [ <logicalconnective> <qlformula> ]
<membership>          ::= identifier ‘∈’ <dotaddress> [ <recursionexpr> ]
                      |   identifier ‘=’ <dotaddress>
<recursionexpr>       ::= ‘@’ ‘.’ <dotaddress>
<logicalconnective>   ::= ‘∧’ | ‘⇔’ | ‘⇒’ | ‘∨’
```

8.3 Predicate Syntax

```
<predicate>           ::= <lhs> ‘==’ <predicaterhs>
<predicaterhs>        ::= [ <quantificationlist> ] [ <plformula> ]
<quantificationlist>  ::= <quantification> [ ‘,’ <quantificationlist> ]
<quantification>      ::= <quantifiersymbol> identifier ‘:’ <compidentifier>
<compidentifier>      ::= identifier [ ‘.’ <dotaddress> ] [ <recursionexpr> ]
<plformula>           ::= <lexpr> [ <logicalconnective> <plformula> ]
<lexpr>               ::= <expr> <relop> <expr>
                      | <pinvocation>
<expr>                ::= <compidentifier> [ <prime> ] [ <arithop> <expr> ]
                      |   <compidentifier> [ <prime> ] [ <setop> <expr> ]
<setop>               ::= ‘∪’ | ‘∩’
<arithop>             ::= ‘+’ | ‘-’ | ‘*’ | ‘/’
<pinvocation>         ::= identifier ‘(’ <arglist> ‘)’
<arglist>             ::= identifier [ ‘,’ <arglist> ]
<quantifiersymbol>    ::= ‘∀’ | ‘∃’
<prime>               ::= ‘′’
<relop>               ::= ‘=’ | ‘≤’ | ‘≥’ | ‘≠’ | ‘>’ | ‘<’
```

9.0 Add refs

[Mal82][Pau90] [Pet81] [Dye89]