Computer Science
Technical Report

Colorado State University

Describing a Top-Down Architectural Style: the PARSE Process Graph Notation

Michael D. Rice
Computer Science Group
Mathematics Department
Wesleyan University

mrice@uts.cs.wesleyan.edu

Stephen B. Seidman
Department of Computer Science
Colorado State University

seidman@cs.colostate.edu

September 17, 1996

Technical Report CS-96-123

Department of Computer Science
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5862 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

Describing a Top-Down Architectural Style: the PARSE Process Graph Notation

Abstract: The *ASDL* language represents a formal methodology for modeling the software architectural styles that are used to develop complex computer systems. In previous work, *ASDL* has been used to model a coarse-grain dataflow style used for large-scale signal processing applications. This style incorporates a bottom-up design methodology. In this paper, the expressiveness of *ASDL* is demonstrated by its use to model a architectural style for parallel and distributed systems that incorporates a top-down design methodology. The *ASDL* model provides a valuable basis for understanding the style and for posing questions to its developers.

1. Introduction

In the 1980s, large-scale software systems came to be regarded as collections of modules, with communication among the modules mediated by clearly specified interfaces. While this view proved to be useful for small and medium-scale software systems, it is insufficient to provide the support for global understanding of an industrial-strength software system. One problem has been that the modules and their interconnections represent only the syntax of a large-scale software system, and provide no basis for talking about system semantics. Recently, many researchers have suggested that abstractions are needed that can support consideration of both syntax and semantics of large-scale software systems. These abstractions have come to be called *software architectures* [SG].

This insight has proved to be helpful, but the large variety of software architectures that have been described has tended to make it difficult to draw widely applicable conclusions about the structure of large software systems. Recently, David Garlan and his associates have suggested that software architectures can be classified as belonging to distinct *architectural styles* [AAG], and that it might be far more efficient to study these styles. The study of software architectural styles is still in its infancy. The first papers from Garlan's group dealt with only a few rather general styles: the pipes and filters commonly used in Unix; the client-server style; and an object-oriented style ([SG], [G2], [GAO]). While all three styles are familiar and widely used, they are all rather general, and none can really be regarded as an industrial-strength style.

We have developed a formalism (*ASDL*) for describing architectural styles, and we have applied it to model the Processing Graph Method (*PGM*) style [RS3]. *PGM* [KS] is a coarse-grained dataflow style developed at the U.S. Naval Research Laboratory to support designing signal processing architectures. An architecture designed under *PGM* can be regarded as a graph whose nodes correspond to signal processing computations (transforms, filters, convolutions, etc.); the edges are used for data communication, and the graph executes in a data-driven fashion. Since an entire graph can be encapsulated in a single node that can be used as a component in other graphs, *PGM* can be characterized as a bottom-up software architectural style.

The expressiveness of *ASDL* will be demonstrated here by showing that it can also be used to describe a top-down architectural style. The Parallel Software Engineering (*PARSE*) [GJGC] methodology was developed to support the top-down design and implementation of parallel and distributed software architectures. *PARSE* has been used to describe a parallel logic language run-time support system [JGG], a parallel database engine [GGJ], a parallel transport protocol for high speed networks [GJGC], and a real time embedded control system [LSG]. In this paper, we will show that *PARSE* can be regarded as an architectural style that can be modeled naturally by *ASDL*.

2. The PARSE architectural style

Architectures developed within *PARSE* are represented graphically as diagrams constructed according to the rules of the *PARSE* Process Graph Notation (*PGN*) [G1]. We will use *ASDL* to represent the entities and rules of *PGN*, and thus to model the *PARSE* methodology.

A *PARSE* architecture is constructed from *communication entities* and *processing entities*. Processing entities consist of *process objects* (further distinguished as *data servers*, *function processes*, and *control processes*) and *external interface objects*. Process objects may have persistent state (data servers, control processes) or be stateless (function processes); at the same time, they may also be active (control processes) or passive (function processes, data servers). *PARSE* also makes a distinction between *classes* and *instances* of process objects. A designer of a *PARSE* architecture first defines classes of process objects and then creates instances of such classes.

Communication entities consist of *paths* and *constructors*. Paths are used to transmit data between instances of process objects. The semantics of data transmission are defined by the assignment of *path types* to paths. *PGN* uses the following path types: *synchronous*, *asynchronous*, *bidirectional*, *broadcast*. Paths are attached to object instances at *ports*. The ports of an object form part of the object's class definition.

The following example will illustrate some of the features of PARSE; it is a simplified version of an example appearing in [GJGC]. Figure 1 illustrates a PARSE description of the high-level design of a system that implements the packet management component of a new transport protocol for broadband networks [CG].

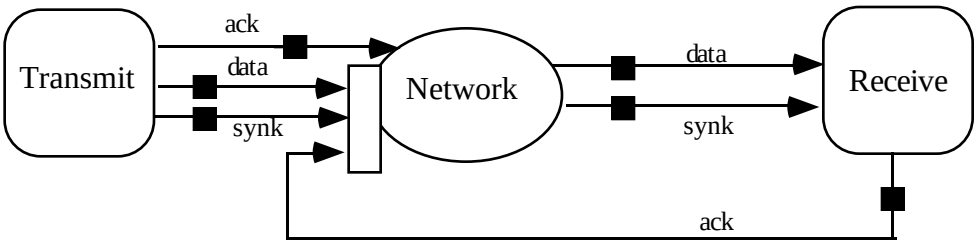

Figure 1. Top-level *PARSE* system description

In Figure 1, ovals represent function servers (*Network*), and rectangles with rounded corners represent control processes (*Transmit*, *Receive*). Communication is indicated by directed arrows; a box on an arrow indicates that the communication is asynchronous. The clear rectangular box at the left side of *Network* indicates a nondeterministic path constructor.

Each process in a top-level *PARSE* system description can be replaced by a system containing lower-level processes that represent explicit design decisions. Figure 2 shows how the *Receive* process of Figure 1 can be decomposed into lower-level processes. An arrow without a box (*Status*) indicates that the communication is synchronous. Circles at the end of arrows indicate that data is to be communicated to or from the exterior of a process; the three arrows terminated with circles correspond to the three external communications with the *Receive* process shown in Figure 1.

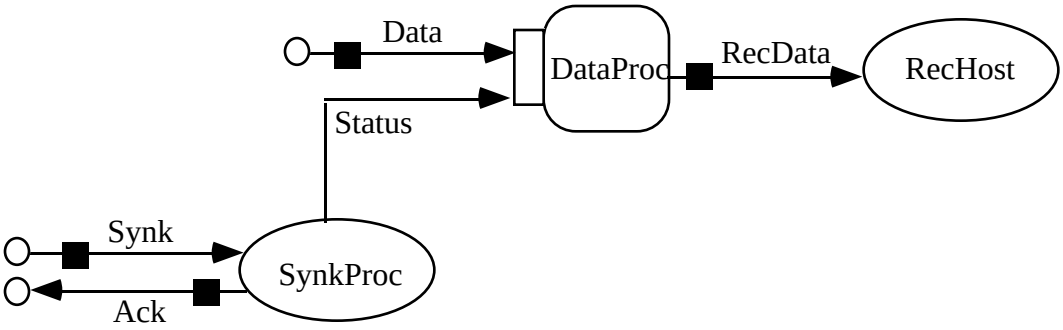

Figure 2. Internal Structure of *Receive*

Both the graphical syntax and the semantics of *PGN* are described informally in [G1]. The *PARSE* literature contains neither a formal description of the syntax nor a way to describe the semantics of a system modeled in *PARSE* (although [RSJC] shows how *PARSE* processes can be translated into Petri nets). In this paper, we will show how the *ASDL* formalism can be used both to represent *PGN* syntax and to assign semantic interpretations to *PARSE* models.

3. Modeling *PARSE* in *ASDL*

ASDL is a generalization of the formalism that was used in [RS1] to describe module interconnection languages (*MIL*s). Such languages express the structure of a software system in terms of constraints imposed on the system's modules and module interfaces. In the *MIL* formalism, *Z* schemas were used to describe these constraints. Generic schemas were used to describe the features that are common to all *MIL*s; a specific *MIL* was modeled by setting the values of generic schema parameters and adjoining application-specific signature elements and constraints. Since the modules and their interfaces can be regarded as representing the syntax of a software architecture, the *MIL Z* schemas provide an initial substrate for *ASDL*. This substrate must be extended to enable it to describe the semantics of the collection of modules making up a software architecture, and the *MIL* schemas have accordingly been extended in *ASDL*.

3.1 The *semantic library* type

The schemas use infinite sets *Labels*, *Nodes*, *Ports*, and *Templates* that are assumed to be disjoint. The **ASDL_Library** schema specifies a *semantic library type* that provides a collection of templates and information about their interfaces. A template represents a computational component, and its interface consists of ports that are used for sending and receiving data. The mapping *interfaces* associates templates with ports. Each port has a set of attributes whose values represent the direction of data movement, the type of the data, and application-dependent information; the generic parameters *Indices* and *Attributes* are used to provide application-specific information about attributes and their values. The attribute values are assigned to ports by the mapping *port-attr*. Templates are also assigned to application-specific categories by the mapping *part*, and *interp* provides semantic interpretations for the templates that are used to construct a specific architecture. The semantic interpretations are members of a set *Interpretations* of *CSP* [H] process expressions. See [RS2] for a formal definition of this set.

Some templates are identified as *primitive templates*; these correspond to software system components that have been preloaded into the library. The members of *Collection* \ *Primitives* are templates that correspond to encapsulated composite modules. Furthermore, members of

Templates \ *Collection* can also serve as *reference templates*, which correspond to interfaces designed in a top-down fashion.

ASDL_Library [Indices, Attributes, Parts]

interfaces : Templates $\rightarrowtail\!\!\!\!\!\!\rightarrow \mathbb{F}_1$Ports	
port-attr : Ports $\rightarrow\!\!\!\!\!\!\rightarrow$ Indices $\rightarrow$ Attributes	
Collection : $\mathbb{F}_1$Templates	
part : Templates $\rightarrow\!\!\!\!\!\!\rightarrow$ Parts	
interp : Templates $\rightarrow\!\!\!\!\!\!\rightarrow$ Interpretations	
Primitives $\subseteq$ Collection	
Collection = *dom* interfaces	
dom interp = *dom* part = Collection	
disjoint ran interfaces	
dom port-attr = $\cup$ *ran* interfaces	
$\forall p \in$ *dom* port-attr $\bullet$ port-attr(*p*).*dir* $\in$ {*in*, *out*}	

In the *ASDL* model of *PARSE*, each class of process objects corresponds to a library template. We will need to augment **ASDL_Library** by adding some signature elements and constraints that are specific to *PARSE*.

PARSE_Library [Indices, Attributes, Parts, Interps]

ASDL_Library [Indices, Attributes, Parts]	
path-constructors : $\mathbb{F}(\mathbb{F}_1$Ports)	
constructor-names : $\mathbb{F}$Ports $\rightarrow\!\!\!\!\!\!\rightarrow$ Char*	
class-names : Templates $\rightarrowtail\!\!\!\!\!\!\rightarrow$ Char*	
PARSE_Library_Constraints	

A *PGN* path constructor corresponds to a nonempty set of ports, and the set *path-constructors* consists of the path constructors that are used in a specific *PARSE* architecture. The functions *constructor-names* and *class-names* are used to assign names to path constructors and to classes of process objects.

We will need to make the following *PARSE*-specific assumptions about the *ASDL* generic parameters:

- *Indices* contains the fields *category*, *name*, *protocol*, *comm-type*, and *constr-type*. The value of a port's *category* and *name* attributes are the category and name of the port's containing process object; the value of its *protocol* and *comm-type* attributes are the protocol and communication type of the port. The value of a port's *constr-type* attribute is the type of the path constructor containing the port.

- *Attributes* contains the names of *PARSE* types

- *Parts* = {*function*, *data_server*, *control_process*, *external_interface*} contains the categories of *PARSE* process objects.

- *Communication_Types* = {*synchronous*, *asynchronous*, *bidirectional*, *broadcast*} contains the *PARSE* communication types.

- *Constructor_Types* = {*concurrent*, *deterministic*, *nondeterministic*, *nil*} contains the *PARSE* constructor types. We assume that every port is contained in a path constructor, which may consist of a single port. In the latter case, the constructor type *nil* is used.

PARSE_Library_Constraints is the conjunction of thirteen individual constraints, listed and annotated below. Each constraint corresponds to one or more features of *PARSE* or *PGN*, and references are given to the *PGN* rules of [G1]. A listing of the rules is given in Appendix A.

(1) *disjoint* path-constructors

The path constructors are disjoint sets of ports ([G1], p. 12).

(2) $\cup$ path-constructors = *dom* port-attr

Each port of a template corresponding to a *PARSE* process object belongs to a path constructor. In [G1], path constructors are only used to link multiple input ports. Since we constrain output ports to belong to singleton path constructors (see (8), (9) below), our model is consistent with *PGN*.

(3) *dom* constructor-names = path-constructors

All path constructors are named. ([G1], PA-07)

(4) *dom* class-names $\supseteq$ Collection

All primitive templates and templates representing encapsulated modules must be named. Note that templates representing top-down modules will also be named; these templates are included in Collection.

(5) $\forall \tau \in$ Collection • part(τ) = *external_interface* $\Rightarrow \tau \in$ Primitives

Templates representing external interface process objects have no internal structure. ([G1], PT-01)

(6) $\forall \tau \in$ Collection, $\forall p \in$ interfaces(τ) •

(port-attr(p).*category* = part(τ)) $\wedge$
port-attr(p).*name* : Char $\wedge$
port-attr(p).*protocol* $\in$ *Protocols* $\wedge$
port-attr(p).*comm-type* $\in$ *Communication_Types*) $\wedge$
port-attr(p).*constr-type* $\in$ *Constructor_Types*)

Port attributes are constrained to take on appropriate values. Since the structure of protocols is application-dependent, the set *Protocols* will not be discussed further here.

(7) $\forall \tau \in$ Primitives, $\forall p \in$ interfaces(τ) • part(τ) = *external_interface*
$\Rightarrow$ port-attr(p).*constr-type* = *nil*

Ports of templates representing external interface process objects must belong to trivial (singleton) path constructors. ([G1], PA-01)

(8) $\forall\ p\ \in$ *dom* port-attr • (port-attr(p).*dir* = *out* $\Rightarrow$ port-attr(p).*constr-type* = *nil*)

Output ports of templates must belong to trivial (singleton) path constructors. [G1, PA-02]

(9) $\forall\ p\ \in$ *dom* port-attr • port-attr(p).*constr-type* $\neq$ *nil*

$$\Leftrightarrow \exists\ \alpha \in \text{path-constructors} \bullet p \in \alpha \wedge \#\alpha > 1$$

Nontrivial path constructors include more than one port. ([G1], PA-02)

(10) $\forall\ p\ \in$ *dom* port-attr • port-attr(p).*constr-type* = *concurrent* $\Rightarrow$

$$p \in \text{interfaces(Collection} \setminus \text{Primitives)}$$

A port of a primitive template cannot belong to a concurrent path constructor. ([G1], PA-03)

(11) $\forall\ \alpha\ \in$ path-constructors •

$$\{p, q\} \subseteq \alpha \Rightarrow \left(\begin{array}{ll} \text{port-attr}(p).\textit{dir} = \text{port-attr}(q).\textit{dir} & \wedge \\ \text{port-attr}(p).\textit{type} = \text{port-attr}(q).\textit{type} & \wedge \\ \text{port-attr}(p).\textit{comm-type} = \text{port-attr}(q).\textit{comm-type} & \wedge \\ \text{port-attr}(p).\textit{protocol} = \text{port-attr}(q).\textit{protocol} & \wedge \\ \text{port-attr}(p).\textit{constr-type} = \text{port-attr}(q).\textit{constr-type}) & \end{array}\right.$$

Ports belonging to the same path constructor must have identical attributes.

Semantic interpretations of templates representing PARSE object classes require some additional formalism. The symbol Λ will denote a nonempty finite set whose elements correspond to ports. A port associated with a template τ will be denoted by (p, τ), where $p \in \Lambda$. The elements of Λ will also be used to refer to *CSP* channels that are used to express the semantics of port communication. If the port semantics require bidirectional communication, this will be expressed by the pair of *CSP* channels $\{ p, p^{\text{op}}\}$, where $p \in \Lambda$.

(12) interfaces(τ) = $(\cup\{\Lambda(\tau, c) : c \in \textit{Communication\_Types}\ )\} \times \{\tau\}$, where

$$\Lambda(\tau, c) = \{p \in \Lambda : \text{port-attr}(p, \tau).\text{comm-type} = c\} \wedge$$

$$\begin{array}{rl} \textit{interp}(\tau)) = & (\|\ {}^{*}\text{Synch}(p)\text{: } p \in \Lambda(\tau, \textit{synchronous})) \ \| \\ & (\|\ {}^{*}\text{Bidir}(p)\text{: } p \in \Lambda(\tau, \textit{bidirectional})) \ \| \\ & (\|\ {}^{*}\text{Asynch}(p)\text{: } p \in \Lambda(\tau, \textit{asynchronous})) \ \| \\ & (\|\ {}^{*}\text{Broadcast}(p)\text{: } p \in \Lambda(\tau, \textit{broadcast})) \end{array}$$

where Synch(p) = Asynch(p) = Broadcast(p) =

if port-attr(p, τ).dir = *in*
then p ? $x \rightarrow$ SKIP
else p ! $x \rightarrow$ SKIP

Bidir(p) = **if** port-attr(p , τ).dir = *in*
then p? $x \rightarrow p^{op}$? $x \rightarrow$ SKIP
else p ! $x \rightarrow p^{op}$! $x \rightarrow$ SKIP))[1]

The CSP process *interp*(τ)) describes the interface semantics of the template τ by using the communication path type associated with a port to specify the communications on the corresponding channels. These channels are always ready to communicate, and they execute concurrently. It is important to note that *ASDL* uses the *Z* and *CSP* formalisms in an orthogonal fashion. The *interp* field of an instance of the **ASDL_Library** schema and the *comp-expr* field of the **ASDL_Setting** schema contain character strings that can be interpreted as *CSP* expressions. Since the goal of *ASDL* is to have *Z* and *CSP* reinforce each other, there is no need to propose a common semantic domain for the two formalisms.

Example: We will use the *Receive* process of Figure 2 to illustrate the semantic library type. For this example, we will need templates and ports satisfying the following properties:

$$\{DataProc, SynkProc, RecHost\} \subseteq Primitives$$

and

$$\{\alpha, \beta, \gamma, \delta, \varepsilon, \eta, \theta\} \subseteq Ports$$

These templates and ports are illustrated in Figure 3. Ports belonging to nontrivial path constructors are indicated by underlines.

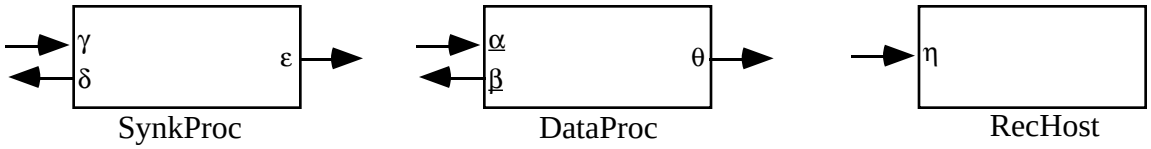

Figure 3. Semantic library elements corresponding to the *Receive* process

The path constructors consist of the sets {α, β}, {γ, δ}, {ε}, {η}, and {θ}; the *port-attr* mapping is defined by the following table:

[1] The symbols || and * are taken from [H]. If P is a set of processes, || P denotes the concurrent execution of the members of P. If P is a process, then *P denotes a process that iterates P without termination.

port	dir	name	category	constr-type	comm-type
α	in	DataProc	control process	nondeterministic	asynchronous
β	in	DataProc	control process	nondeterministic	synchronous
γ	in	SynkProc	function server	nil	asynchronous
δ	out	SynkProc	function server	nil	asynchronous
ε	out	SynkProc	function server	nil	synchronous
η	in	RecHost	function server	nil	asynchronous
θ	out	DataProc	control process	nil	asynchronous

3.2 The *semantic module* type

The **ASDL_Setting** schema, given below, specifies a *semantic module type* that is based on the library type. This type consists of a set of *nodes* and a set of *connections* that are defined by *shared labels*. Each node is instantiated from a library template. The ports on an instantiated node are called *slots*. A label shared by two or more slots creates a connection that can be used for data movement between the corresponding nodes.

ASDL also contains operations that support the construction of a specific software architecture within an architectural style described by the schema constraints. For example, the *Create_Node* operation instantiates a node from a library template, and the *Assign_Label* operation assigns a label to an unlabeled slot. Schemas defining these operations are found in [RS2].

The semantic module type also contains a *composition expression* that specifies how the nodes in a setting are composed for execution purposes and a *semantic description* mapping that assigns a semantic abbreviation to each label used in a module. A node instantiated from a reference template is called a *pseudonode*. A composition expression is a restricted type of *timed CSP* process, in which node names are viewed as processes. For example, the expression may specify that the nodes in a setting will be executed in parallel. The sets *ProcessExpressions* and *SemanticDescriptions* are described in [RS2].

The execution semantics of an *ASDL* module are derived from the semantic interpretations of the templates underlying the nodes, the composition expression, and the semantic abbreviations associated with the labels that specify the connections between nodes. *ASDL_Setting* therefore contains all of the information needed to execute the module.

It has already been mentioned that *ASDL* models of software architectural styles use semantic abbreviations assigned to shared labels to describe the way in which data is communicated between nodes. If this approach were applied directly to *PGN*, data communication between process objects would be expressed in terms of attributes of these objects and their ports. *PGN* actually uses an alternative approach; data communications between objects are determined by path constructors associated with ports of the underlying object classes. A faithful model of *PGN* (and *PARSE*) in *ASDL* must reflect this approach. In order to do so, we will introduce the specific semantic abbreviation **nil**. The semantics of a slot label mapped to **nil** by *semantic-descr* will be inherited from the interface semantics of the corresponding port on the underlying template, and the execution semantics of a *PGN* (*PARSE*) module will therefore be appropriately derived from the composition expression and the semantic interpretations of the templates underlying the nodes.

ASDL_Setting [Indices, Attributes, Parts, SemanticDescriptions]

ASDL_Library [Indices, Attributes, Parts]		
node-parent	:	Nodes $\rightarrow\!\!\!\!\!\mid$ Templates
slots	:	$\mathbb{F}$(Nodes $\times$ Ports)
slot-attr	:	Nodes $\times$ Ports $\rightarrow\!\!\!\!\!\mid$ Indices $\rightarrow$ Attributes
label	:	Nodes $\times$ Ports $\rightarrow\!\!\!\!\!\mid$ Labels
comp-expr	:	ProcessExpressions
semantic-descr	:	Labels $\rightarrow\!\!\!\!\!\mid$ SemanticDescriptions

slots = *dom* slot-attr
dom label $\subseteq$ slots
$\forall n \in$ *dom* node-parent •
node-parent(n) $\in$ Collection $\wedge$ $p \in$ interfaces(node-parent(n))
$\Rightarrow$ (n, p) $\in$ *dom* slot-attr $\wedge$ slot-attr(n, p) = port-attr(p)
dom semantic-descr = *ran* label

In the *ASDL* model of *PARSE*, an instantiation of a process object corresponds to a node. We will need to modify **ASDL_Setting** by adding some signature elements and constraints that are specific to PARSE. The functions *node-name* and *state* are used to assign names to nodes and to record the internal states of nodes. The domain **D** used to represent node state is application-dependent, and will not be discussed further here.

PARSE_Setting [Indices, Attributes, Parts, SemanticDescriptions]

ASDL_Setting [Indices, Attributes, Parts, SemanticDescriptions]
PARSE_Library
node-name : Nodes $\rightarrow\!\!\!\!\!\mid$ Char*
state: Nodes $\rightarrow\!\!\!\!\!\mid$ **D**

PARSE_Setting_Constraints

PARSE_Setting_Constraints is the conjunction of the five individual constraints that are listed below. Each constraint corresponds to one or more features of *PARSE* or *PGN*, and explanations of the correspondence are given in boldface, with cross-references to [G1]. *comp-expr* uses *CSP* channels to describe communication between ports corresponding to (input, output) slot pairs that share a common label. These channels will be designated by the labels to which they correspond. If bidirectional communication is needed, a pair of channels $\{a, a^{op}\}$ will correspond to the label *a*.

(1) *dom* node-name = *dom* node-parent

All PARSE process objects must be named.

(2) ($\forall$ s, t $\in$ slots • (label(s) = label(t)) $\Rightarrow$
(slot-attr(s).*protocol* = slot-attr(t).*protocol*) $\wedge$
(slot-attr(s).*type* = slot-attr(t).*type*) $\wedge$
(slot-attr(s).*comm-type* = slot-attr(t).*comm-type*))

Since shared slot labels represent communication paths, slots with the same label must correspond to ports that have the same type, protocol, and communication type ([G1], p. 7).

(3) $(\forall\ s, t \in \text{slots}) \bullet$

$((\text{label}(s) = \text{label}(t)) \wedge (\text{first}(s) = \text{first}(t)) \Rightarrow$

$\exists\ \alpha \in \text{path-constructors} \wedge \{\text{second}(s), \text{second}(t)\} \subseteq \alpha\ )$

The projection functions *first* and *second* map a slot $s = (n, p)$ to n and p, respectively. Since shared slot labels represent communication paths, slots on the same node with the same label must correspond to ports that belong to the same path constructor.

(4) $(\forall\ n \in \text{Nodes}) \bullet n \in \textit{dom}\ \text{state} \Rightarrow$

$n \in \textit{dom}\ \text{node-parent} \wedge \text{part}(\text{node-parent}(n) \in \{\textit{data-server}, \textit{control-process}\}$

Only data server processes and control processes have persistent state ([G1], pp. 2-3).

(5) comp-expr = || {Proc(n) | $n \in$ *dom* node-parent}

The processes corresponding to the nodes of a PARSE graph execute concurrently.

Proc(n) = Internal-process(n) || Interface-process(n)

The process corresponding to a single node of a PARSE graph represents both the execution semantics (*Internal-process*) and the interface semantics (*Interface-process*) of the node. The execution semantics will not be specified further here.

Interface-process(n) = || {Grouped-slot-process (a) | $a \in$ *ran* label}

The interface semantics of a node are obtained from the concurrent composition of the behavior of the groups of slots that share common labels. The behavior of such a slot group will depend on the type of the path constructor that is associated with the underlying template ports.

Grouped-slot-process (a) =

Operator (slot-attr(s).*constr-type*) {Labeled-slot-process(s) | label (s) = a }

where *Operator* is a function from *Constructor Types* to CSP constructors defined by the ordered pairs

{(*concurrent*, ||), (*determinist[illegible]*, [illegible] (*nond[illegible]nistic*, [illegible] (*nil*, NIL)}

[2] The operator symbols $\Box$ and Π are taken from [illegible]. If [illegible] set of processes, $\Box$ [illegible] Π [illegible] te nondeterministic execution of the members of [illegible] first operator allows the environment to select the process that will be executed.

Since PARSE_Setting_Constraint (3) and PARSE_Library_Constraint (11) imply that all slots *s* with the same label will have the same value of slot-attr(*s*).*constr-type*, the argument of *Operator* is well-defined.

By PARSE_Library_Constraint (9), the constructor type *nil* will only be applied to singletons. The corresponding NIL operator is defined by NIL(P) = P; it will only be applied to a set containing a single *CSP* process.

Labeled-slot-process (*s*) =

if port-attr(*p*).comm-type = *synchronous*
then if port-attr(*p*).dir = *out* **then** $*(p?x \rightarrow a!x \rightarrow \text{SKIP})$
else $*(a!x \rightarrow p?x \rightarrow \text{SKIP})$

else if port-attr(*p*).comm-type = *bidirectional*
then if port-attr(*p*).dir = *out* **then** $*(p?x \rightarrow a!x \rightarrow a^{op}?x \rightarrow p^{op}!x \rightarrow \text{SKIP})$
else $*(a?x \rightarrow p!x \rightarrow p^{op}?x \rightarrow a^{op}!x \rightarrow \text{SKIP})$

else if port-attr(*p*).comm-type = *asynchronous*
then if port-attr(*p*).dir = *out* **then** $*(p?x \rightarrow a^{in}!x \rightarrow \text{SKIP}) \,||\, \text{Buff}(a)$
else $*(a^{out}?x \rightarrow p!x \rightarrow \text{SKIP}) \,||\, \text{Buff}(a)$,

else if port-attr(p).comm-type = *broadcast*
then if port-attr(p).dir = *out* **then** $*(p?x \rightarrow (\|a_{\alpha}!x \rightarrow \text{SKIP} \mid \text{slot-attr}(\alpha).\text{dir} = in \wedge \text{label}(\alpha) = a)$
else $(a?x \rightarrow p!x \rightarrow \text{SKIP})$

In *Labeled_slot_process*,

(1) *a* = label(*s*) is the label associated with the slot *s* and *p* = second(*s*) is the corresponding port

(2) Buff(*a*) is a buffer process of infinite capacity served by channels a^{in} and a^{out}.

Example: The *Create_Node* operation can be called three times to use the templates given above to construct the system of Figure 2. If this is done, we will have

$$\{dataproc\_node, synkproc\_node, rechost\_node\} \subseteq Nodes,$$

$$node\text{-}parent\ (dataproc\_node) = DataProc$$
$$node\text{-}parent\ (synkproc\_node) = SynkProc$$
$$node\text{-}parent\ (rechost\_node) = RecHost,$$

$$\{(DataProc, \alpha), (DataProc, \beta), (DataProc, \theta), (SynkProc, \gamma),$$
$$(SynkProc, \delta), (SynkProc, \varepsilon), (RecHost, \eta)\} \subseteq Slots,$$

and the mapping *slot-attr* will be defined, as specified in the **PARSE_Setting** schema, by

slot-attr((*DataProc*, α) = port-attr(α), etc.

The *Assign_Label* operation can then be used to set the values of the slot labels, so that we will now have

label(*DataProc*, θ) = *label*(*RecHost*, η) = *RecData*

and

label(*DataProc*, β) = *label*(*SynkProc*, ε) = *Status*

Figure 4 illustrates the ASDL model of the system of Figure 2. Rectangles correspond to nodes, while slots and labels are indicated by internal and external annotations, respectively.

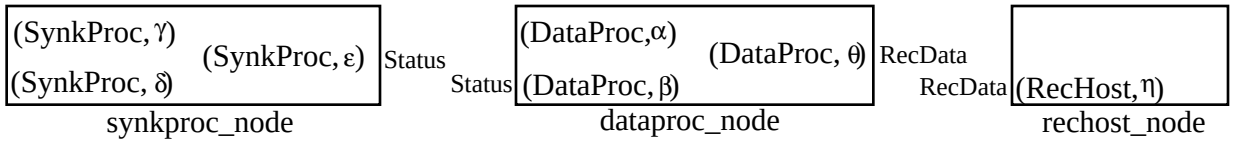

Figure 4. Examples of ASDL semantic module types

3.3 Unit types

The **ASDL_Setting** schema represents a module as a self-contained computational unit without any external connections. The **ASDL_Unit** schema corresponds to a *unit type* that describes these connections and the associated interface semantics. It includes a set of *virtual ports* that represent the "public" interfaces of the unit and a mapping that specifies the attributes of these ports. The mapping *virtual-port-descr* assigns a semantic abbreviation to each virtual port in a unit. The virtual ports and their attributes, specified in the associated **ASDL_Boundary** schema, represent a unit's *syntactic boundary*. The *connect* mapping describes the links between slots and virtual ports.

In *ASDL*, units can be used in two rather different ways. A unit can represent the boundary of an existing setting. The association between the setting's slots and the unit's virtual ports is expressed by the *connect* mapping. Alternatively, an "empty" unit can consist of the virtual ports that comprise a boundary. The *Create_Unit_Node* operation adds a node based on such a unit to a setting and creates slots corresponding to the unit's virtual ports. A top-down model of a system can be constructed in this way The *Connect_Virtual_Port* operation can then be used to refine the top-down model by filling in the internal structure of the nodes instantiated from the empty units. Schemas describing these operations can be found in [RS2].

Example: These concepts can be illustrated with the transport protocol example shown in Figure 1. A top-down model of this system initializes *Units* to {*Transmit*, *Network*, *Receive*}. The virtual ports are given as follows:

Transmit: a, b, c

Network: d, e, f, g, h, i

Receive: j, k, l

and $\{a, b, c, d, e, f, g, h, i, j, k, l\} \subseteq Ports$.

The path constructors consist of the sets $\{e, f, g\}, \{j, k\}, \{a\}, \{b\}, \{c\}, \{g\}, \{l\}\}$. The *port-attr* mapping is defined by the following table:

Port	dir	name	category	constr-type	comm-type
a	in	Transmit	control process	nil	asynchronous
b	out	Transmit	control process	nil	asynchronous
c	out	Transmit	control process	nil	asynchronous
d	out	Network	function server	nil	asynchronous
e	in	Network	function server	nondeterministic	asynchronous
f	in	Network	function server	nondeterministic	asynchronous
g	in	Network	function server	nondeterministic	asynchronous
h	out	Network	function server	nil	asynchronous
i	out	Network	function server	nil	asynchronous
j	in	Receive	control process	concurrent	asynchronous
k	in	Receive	control process	concurrent	asynchronous
l	out	Receive	control process	nil	asynchronous

If the *Create_Unit_Node* operation is invoked once for each of the three units, we will then have $\{trans\_node, net\_node, rec\_node\} \subseteq Nodes$, $\{t, u, v\} \subseteq Templates$, and $\{(trans\_node, t), (net\_node, u), (rec\_node, v)\} \subseteq$ *node-parent*. We will also have

$$Slots = \{t\} \times \{a, b, c\} \cup \{u\} \times \{d\ e, f, g, h, i\} \cup \{v\} \times \{j\ k, l\}$$

The values of the *slot-attr* mapping will be defined explicitly from the corresponding values of *port-attr*. It should be noted that while *slot-attr* must be defined explicitly for nodes obtained from units in a top-down design process, the **ASDL_Setting** schema defines it implicitly for nodes obtained from templates as part of bottom-up design.

The *Assign_Label* operation can then be used to make label assignments that will create the connections shown in Figure 1:

label(t, a) = label (u, d) = *ack1*
label(t, b) = label (u, e) = *data1*
label(t, c) = label (u, f) = *synk1*
label(u, g) = label (v, l) = *ack2*
label(u, h) = label (v, j) = *data2*
label(u, i) = label (v, k) = *synk2*

The refinement of the system of Figure 1 by the system of Figure 2 can then be accomplished by calling the *ConnectVirtualPort* operation, which will give the *connect* function the following values:

$$\text{connect}(\mathit{DataProc}, \alpha) = j$$
$$\text{connect}(\mathit{SynkProc}, \gamma) = k$$
$$\text{connect}(\mathit{SynkProc}, \delta) = l$$

Figure 5 illustrates these concepts. Large rectangles correspond to unit nodes; slots and their labels are indicated by annotations inside and outside these rectangles, respectively. Slots corresponding to ports contained in nontrivial path constructors are indicated by underlines. The refinement of *rec_node* is indicated by the inclusion of squares corresponding to the templates*DataProc* (DP), *SynkProc* (SP), and *RecHost* (RH). The association of slots on unit nodes with slots of the system of Figure 4 is shown by dotted lines. Internal associations between these slots are not shown.

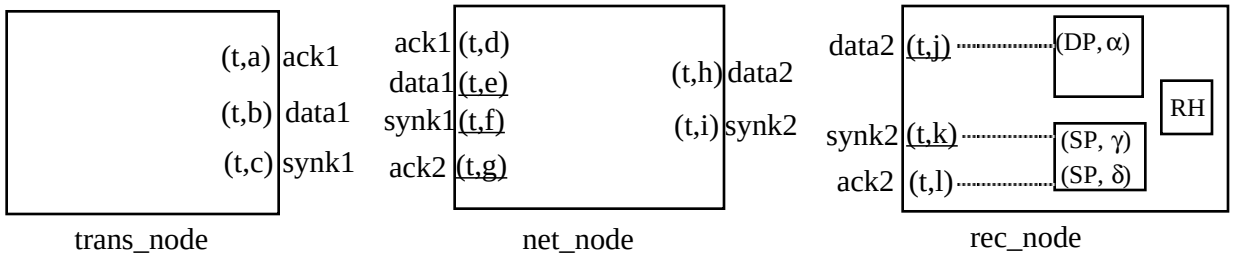

Figure 5. Unit nodes and refinement in ASDL

The **ASDL_Boundary** and **ASDL_Unit** schemas define the *ASDL* unit type.

ASDL_Boundary [Indices, Attributes]

interface-attr : Ports $\rightarrow\!\!\!\!\!\mid$ Indices $\rightarrow$ Attributes virtual-ports : $\mathbb{F}$Ports
virtual-ports = *dom* interface-attr

ASDL_Unit [Indices, Attributes, Parts, SemanticDescriptions]

ASDL_Setting [Indices, Attributes, Parts, SemanticDescriptions] **ASDL_Boundary** [Indices, Attributes] connect : Nodes $\times$ Ports $\rightarrow\!\!\!\!\!\mid$ $\mathbb{F}$Ports virtual-port-descr : Ports $\rightarrow\!\!\!\!\!\mid$ SemanticDescriptions
dom connect $\subseteq$ slots $\cup$ *ran* connect $\subseteq$ virtual-ports *dom* virtual-port-descr = virtual-ports $\forall p \in$ virtual-ports $\bullet$ {interface-attr(p).dir} = {slot-attr(s).dir : $p \in$ connect(s)}

In order to use the *ASDL* unit type in the *PARSE* model, two additional constraints are needed. The **PARSE_Unit** schema includes the *PARSE* versions of the Setting and Library schemas, and also adjoins *PARSE_Unit_Constraints*, which consists of the conjunction of the two additional constraints.

PARSE_Unit
ASDL_Unit [Indices, Attributes, Parts, SemanticDescriptions] **PARSE_Setting** **PARSE_Library**
PARSE_Unit_Constraints

The two additional constraints are:

(1) $\forall(n, p) \in dom$ connect • # connect $(n, p) = 1$

The generality of the *ASDL* unit type allows a slot to be connected to more than one virtual port; this connection pattern is not possible in *PARSE*.

(2) $\cup ran$ connect = virtual-ports

ASDL allows virtual ports that are connected to no slots; this connection pattern is not possible in *PARSE*.

3.4 System Type

The **ASDL_System** schema defines a *system type* that provides all of the structural information needed to understand a software architecture constructed in the modeled style. The schema includes the **ASDL_Library** schema, as well as architectural state information: the modules and units that have been used to describe the bottom-up and top-down components of an architecture, the relationships between units and modules, and the connection between library templates and unit types. *ASDL* also supports the representation of multiple logical views of a software architecture, and the **ASDL_System** schema correspondingly includes information on architectural connections between different logical views.

The schema variable *Units* denotes the set of all unit types that have been used to describe various logical views of the architecture. The *basis* mapping summarizes the connection between unit types and templates. Finally, the *relation* mapping summarizes the relations that have been specified between sets of nodes in various units to provide architectural connections between different logical views.

In order to represent the structure of a *PARSE* architecture, it will be necessary to modify the **ASDL_System** schema to use the *PARSE*-specific versions of the *ASDL* types, to add a function that allows objects to be named, and to adjoin three *PARSE*-specific system constraints. The modified schema is given below.

ASDL_System [Indices, Attributes, Parts, SemanticDescriptions]

ASDL_Library [Indices, Attributes, Parts, SemanticDescriptions]
Units : 𝔽**ASDL_Unit** [Indices, Attributes, Parts, SemanticDescriptions]
basis : Templates —|→ **ASDL_Unit** [Indices, Attributes, Parts, SemanticDescriptions]
relation : Labels —‖→ 𝔽(Nodes × N
Collection \ *dom* basis = F
Units = *ran* basis
∀ τ ∈ Collection \ Primitiv s(τ).virtual-ports ∧
s(τ) = basis(τ).interface-
∀ ρ ∈ *dom* relation ∃ { ,
(*dom* relation(ρ) ⊆ *dom* .no parent *n* relation(ρ) ⊆ *dom* *.n parent)

PARSE_System [Indices, Attributes, Parts, SemanticDescriptions]

PARSE_Library [Indices, Attributes, Parts, SemanticDescriptions]
Units : 𝔽**PARSE_Unit** [Indices, Attributes, Parts, SemanticDescriptions]
basis : Templates —|→ **PARSE_Unit** [Indices, Attributes, Parts, SemanticDescriptions]
relation : Labels —‖→ 𝔽(Nodes × Nodes)
instance-names : Node ar*
Collection \ *dom* basis = F
Units = *ran* basis
∀ τ ∈ Collection \ Primitiv s(τ).virtual-ports ∧
s(τ) = basis(τ).interface-
∀ ρ ∈ *dom* relation ∃ { ,
(*dom* relation(ρ) ⊆ *dom* .no parent *n* relation(ρ) ⊆ *dom* *.n parent)

PARSE_System_Constraints

PARSE_System_Constraints is the conjunction of the following constraints:

(1) (∀*T* ∈ Collection\Primitives, {*p* , *q*} ⊆ *T*.interfaces •
port-attr(*p*).constructor-type = port-attr(*q*).constructor-type = *concurrent* ⇒
(∃ *s*, *t* ∈ basis(*T*).slots • connect(*s*) = *p* ∧ connect(*t*) = *q*) ∧ first(*s*) ≠ first(*t*))

A concurrent path constructor can only be connected to ports located on different objects. ([G1], PA-04)

(2) (∀*T* ∈ Collection\Primitives, {*p* , *q*} ⊆ *T*.interfaces ∧ *p* ≠ *q* •
port-attr(*p*).constructor-type = port-attr(*q*).constructor-type
∈ {*deterministic, nondeterministic}* ⇒
(∃ *s*, *t* ∈ basis(*T*).slots •
(connect(*s*) = *p* ∧ connect(*t*) = *q* ∧
port-attr(second(*s*)).constructor-type =
port-attr(second(*t*)).constructor-type = port-attr(*p*).constructor-type))

For a decomposible process, pat[redacted]ned by [redacted]determin[redacted] or a nondeterministic path constructor must be joined [redacted]he same [redacted]f constr[redacted] at a lower level of decomposition. ([G1], PA-05)

(3) *dom* instance-names = ∪{*dom* .no[redacted]parent [redacted]s} ∧ [redacted]

ran instance-names ∩ ran class-names = ∅

Every instance of a process object must have a unique name. The constraint also requires that instance names be distinct from class names. ([G1], p. 4)

4. Conclusions

Developing an *ASDL* model for *PARSE* serves several purposes. First, it provides further evidence for the claim made in [RS2] that *ASDL* is a general language for describing software architectural styles. In addition, the model's ability to represent the complex interface semantics supported by *PARSE* demonstrates the expressiveness of *ASDL*'s process expressions.

Even more importantly, the model provides a valuable platform for posing questions about *PARSE* and *PGN*. It is always difficult to ask questions about aspects of a software architectural style if the only tools are those provided by the style itself. An *ASDL* model of a style requires that there has been an extended interaction between the model builders and the style developers. As the model builders seek to determine the values of the model's generic parameters and to specify the style-specific constraints that must be added to the ASDL schemas, they must interact with the style developers. This interaction not only leads to a more faithful model, it often serves to give the developers an improved understanding of their style. The resulting model can provide a sound basis for implementors and users of the style.

References

[AAG] G. Abowd, R. Allan, and D. Garlan, Using style to understand descriptions of software architecture, *Proceedings of ACM SIGSOFT93 Symposium on Foundations of Software Engineering*, pp. 9-20, 1993.

[CG] T. S. Chan and I. Gorton, "A Transputer-based implementation of HTPNET: a transport protocol for broadband networks", in *Transputer Applications and Systems, vol. 2, Proceedings of the 1993 World Transputer Conference*, pp. 899-910.

[G1] J. P. Gray, "Definition of the PARSE process graph notation, version 2", Technical Report PARSE-TR-2b, Department of Computer Science, University of Wollongong, 1994.

[G2] D. Garlan, "What is style", *Proceedings of First International Workshop on Architectures for Software Systems*, 1995.

[GAO] D. Garlan, R. Allen, and J. Ockerbloom, "Exploiting style in architectural design environments", *Proceedings of ACM SIGSOFT94 Symposium on Foundations of Software Engineering*, 1994.

[GGJ] J. P. Gray, I. Gorton, and I. E. Jelly, "Designing parallel database programs using PARSE", *Proceedings of 17th International Software and Applications Conference*, Phoenix, 1993.

[GJGC] I. Gorton, I. E. Jelly, J. P. Gray, and T. S. Chan, "Reliable parallel software construction using PARSE", *Concurrency: Practice and Experience*, 1995.

[H] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[JGG] I. E. Jelly, I. Gorton, and J. P. Gray, "A hybrid transputer based architecture for parallel logic language execution", *Proceedings of Transputer Applications and Systems '93*, pp. 950-964.

[KS] D. Kaplan and R. Stevens, "Processing graph method 2.0 semantics", manuscript, Naval Research Laboratory, June, 1995.

[LSG] A. Y. Liu, T. S. Chan, and I. Gorton, "Designing distributed multimedia systems using PARSE", *Proceedings of the IFIP Workshop on Parallel and Distributed Software Engineering*, Berlin, March 1996.

[RS1] M. Rice and S. Seidman, "A formal model for module interconnection languages", *IEEE Transactions on Software Engineering* **20** (1994), 88-101.

[RS2] M. Rice and S. Seidman, "Using Z as a Substrate for an Architectural Style Description Language", Technical Report CS-96-120, Department of Computer Science, Colorado State University, 1996.

[RS3] M. Rice and S. Seidman, "Describing the PGM architectural style", Technical Report CS-96-123, Department of Computer Science, Colorado State University, 1996.

[RSJC] S. Russo, C. Savy, I. Jelly, and P. Collingwood, "Petri net modelling of PARSE designs", *Proceedings of EuroPar '96*, Springer-Verlag, 1996.

[SG] M. Shaw and D. Garlan, *Software Architecture: An Emerging Discipline*, Prentice-Hall, Upper Saddle River, NJ, 1996.

Appendix A: *PGN* rules: comments and *ASDL* implementations

The rules referenced and annotated here are those contained in the *PARSE-TR-2b* technical report [G1]. The references and remarks in brackets point to specific features of *ASDL* or to constraints given in the *PARSE* schemas.

Rule PT-01 External interface objects are not decomposible. [Library (5)]

Rule PT-02 All process objects are either decomposible processes or primitive processes [Process objects are represented by members of Collection. Templates are either primitive or composite.]

Rule PT-03 Decomposible processes may be decomposed into an arbitrary collection of lower level processes. [*ASDL* places no restriction on the depth of hierarchical structures.]

Rule PA-01 Constructors may be applied to any type of process object, but may not be applied to external interface objects. [Library (7)]

Rule PA-02 A path constructor can only be applied to multiple input path connections on the same process. [Library (9)]

A constructor must be applied to either multiple input paths, or to a single input path connected to a replicated process. [Library (8); so far, the ASDL model does not treat process replication.]

Rule PA-03 A concurrent constructor cannot be attached to a primitive sequential process. [Library (10)]

Rule PA-04 A concurrent constructor implies that there must be a minimum number of concurrent processes at a lower level of decomposition. There must be at least N internal processes within the decomposition, where N is the number of input path connections entering a constructor, and each path connection must be to a different internal process. [System (1)]

Rule PA-05 For a decomposible process, paths joined by either a deterministic or a non-deterministic constructor, must be be joined by the same type of constructor at a lower level of decomposition. The set of paths joined by a constructor at a higher level, must be joined at a single process, with a constructor of the same type, at a lower level of decomposition. [System (2)]

Rule PA-07 Processes may have many constructors attached to them, they may be of the same type or different types, but each constructor must be labelled with an identifier. For convenience and tidiness, a constructor symbol may be copied so long as each copy is clearly marked. [Library (3)]

Rule PA-08 When a process has many constructors attached to it, there is no implied combination of constructors. [The semantics of this rule are unclear.]