Describing software architectures and architectural styles

M. D. Rice
Computer Science Group
Mathematics Department
Wesleyan University
Middletown, CT 06459 USA
Email:mrice@caucus.cs.wesleyan.edu

S. B. Seidman
Department of Computer Science
Colorado State University
Fort Collins, CO 80523 USA
Email: seidman@cs.colostate.edu

1 INTRODUCTION

Software architectures are abstract models that give syntactic and semantic information about the components of software systems and the relationships between those components. Architectural organizing principles for software systems are called *architectural styles.* We have developed a language called ASDL (Rice and Seidman, 1996) for describing and comparing architectures and styles. An ASDL description of a software architecture or style uses three elements: *templates*, *settings*, and *units*. *Templates* represent interfaces of components that are available for inclusion into an architecture. *Settings* represent architectures that have been built by instantiating templates. *Units* represent system hierarchy: they can encapsulate settings or represent interfaces designed in a top-down manner. Each element is associated with a generic Z schema (Spivey, 1989) that represent structural features common to all styles. Features associated with a particular style are expressed by assigning values to the generic parameters and adding declarations, constraints, and operations to the schemas, constraining the configuration of the elements that make up an architecture. ASDL also includes operations that support construction of a software architecture within a style.

2. AN OVERVIEW OF ASDL

2.1 Templates and Settings

The following generic schema describes ASDL templates:

ASDL_Library [Indices, Attributes, Parts]

interfaces	:	Templates $\rightarrowtail\!\!\!\!\!\!\mid\ \mathbb{F}_1$Ports
port-attr	:	Ports $\rightarrowtail\!\!\!\!\!\!\mid$ Indices $\rightarrow$ Attributes
part	:	Templates $\rightarrowtail\!\!\!\!\!\!\mid$ Parts
interp	:	Templates $\rightarrowtail\!\!\!\!\!\!\mid$ Interpretations
Collection	:	$\mathbb{F}_1$Templates

Primitives $\subseteq$ Collection
Collection = *dom* interfaces = *dom* interp = *dom* part
disjoint ran interfaces
dom port-attr = $\cup$ *ran* interfaces
{*dir*, *type*} $\subseteq$ Indices $\wedge$ {*in*, *out*} $\subseteq$ Attributes
$\forall p \in$ *dom* port-attr $\bullet$ port-attr(*p*).*dir* $\in$ {*in*, *out*}

The values of the schema parameters describe a specific style. The templates in *Collection* represent interfaces of components that can be included into a software architecture. The elements of the schema define the *ports* that constitute a template's interfaces. *port-attr* assigns attribute values that determine the characteristics of a port, *part* assigns each template to a style-specific category, and *interp* defines a template's interface semantics as a composition of guarded *CSP* processes (Hoare 1985, Rice and Seidman 1996). If a template τ has ports *p* and *q* with direction attributes *in* and *out*, then the CSP process

$$interp(\tau) = *(p\ ?\ x\ \rightarrow q\ !\ f(x)\ \rightarrow \mathbf{SKIP})$$

specifies that the template acts as a filter represented by the function *f*..

Primitive templates correspond to interfaces of architecture components that have been preloaded into the library. Members of *Collection* \ *Primitives* are templates that correspond to interfaces of encapsulated composite architectures. Members of *Templates* \ *Collection* can serve as *reference templates* that correspond to interfaces designed in a top-down fashion.

The schema constraints define the requirements needed for any style. They require that each template have a nonempty interface, a category assignment, attribute values, and interface semantics, and that the interfaces of distinct templates are disjoint. Furthermore, they require that *dir* and *type*, which give a port's direction and data type, are indices supplied for all styles, and that the only acceptable attribute values for *dir* are *in* and *out*.

Settings represent architectures that have been built by instantiating templates as *nodes* that represent the architecture's components. A node has external interfaces called *slots* that correspond to (and inherit attributes from) ports on the node's underlying template. Slots can be labeled; shared labels are used to represent relationships among nodes, such as data communication.

The following generic schema describes ASDL settings:

ASDL_Setting [Indices, Attributes, Parts]

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
dom semantic-descr = *ran* label
$\forall n \in$ *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)

The schema components define the syntactic features of the nodes representing the components of an architecture: the templates from which the nodes were instantiated, the node slots, and the interfaces' characteristics and labels. Other schema components describe the semantics of the architecture. *Semantic_descr* assigns semantic abbreviations to labels, and *composition_expr* specifies how the nodes in a setting are composed for execution purposes. A composition expression

is a CSP process in which node names are viewed as processes. For example, it may specify that the nodes in a setting will be executed in parallel.

The constraints require that the slots representing the interfaces of a node *n* consist of the pairs (*n*, *p*), where *p* is a port of the template node-parent(*n*), that the slot attributes are inherited from those of the corresponding ports, and that semantic abbreviations are assigned to all labels.

Semantic abbreviations correspond to a variety of communication capabilities. For example, the abbreviation **usc** represents unidirectional synchronous communication; its *meaning* is given by the CSP expression

$$[\![\ \mathbf{usc}\ ]\!] = *(in\ \mathbf{?}\ x \rightarrow out\ \mathbf{!}\ x \rightarrow \mathbf{SKIP}).$$

The execution semantics of a module can be derived from the semantic interpretations of the templates underlying the nodes, the composition expression, and the semantic descriptions of the labels that specify the connections between nodes.

2.2 Units and Hierarchy

The **ASDL_Setting** schema represents an architecture as a self-contained computational unit. The **ASDL_Unit** schema uses *virtual ports* to represent an architecture's external connections. *virtual-port-descr* assigns semantics to virtual ports, and *connect* describes the links between slots and virtual ports.

ASDL_Unit [Indices, Attributes, Parts]

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

ASDL_Unit requires that the direction of data movement for a virtual port be consistent with that of any slots to which it is linked. Further restrictions are style-dependent. For example, type-consistency requirements may be placed on *connect*, and *virtual-port-descr* may specify broadcasting or multiplexing behavior. Since ASDL operations can be used to add the external interface of a unit into the library as a template, units provide powerful and flexible support for hierarchical descriptions of software architectures.

2.3 Operations

ASDL's generic operations support incremental specification of software architectures: *setting operations* to create and delete nodes, assign labels to slots, specify a composition expression, and select semantic abbreviations, *interface operations* to specify virtual ports, attributes, links, and virtual port descriptions, an *encapsulation operation* to create a new library template based on a unit, and operations that define the units needed to support top-down design. Style-specific operations are constructed by adding new signatures and constraints to generic operations or by incorporating operations into a new operation.

For example, the encapsulation operation **ASDL_External** creates a new library template from a unit. The virtual ports of the unit become the ports of the template. The attributes of these ports are derived from the unit's interface. The template's interpretation is derived from the unit's semantic information: template interpretations, composition expression, label abbreviations, and semantics of the virtual ports.

In Rice and Seidman (1994), a family of generic connector templates was used to describe a top-down design methodology. This is easier in ASDL, since it provides an operation to incorporate a unit into an existing setting as a *pseudonode* based on a reference template. The included unit corresponds to an empty setting whose internal structure can be constructed later.

3. EXAMPLE: DESCRIBING THE PGM ARCHITECTURAL STYLE

3.1 Overview of PGM

The *Processing Graph Method* (*PGM*) is a coarse-grain dataflow software methodology developed at the US Naval Research Laboratory for signal processing applications. The nodes of a PGM graph (Kaplan and Stevens, 1995) consist of *PGM transitions* that represent computations and data restructuring operations, and *PGM places* that represent data transfers. Graph edges may only exist between nodes of different category. The execution of a transition is triggered by the arrival of sufficient data at the transition's input ports. The transition then reads data from these ports, performs the specified computation, and writes data to the transition's output ports. Places offer two forms of data transmission between transitions: *queues* use a first-in, first-out communication protocol and *graph variables* provide rewritable data.

A PGM graph corresponds to a software architecture, and the constraints governing the configuration of PGM graphs define a *PGM architectural style*. A textual description of these constraints is given in Kaplan and Stevens (1995).

3.2. Describing the PGM style in ASDL

PGM nodes are either *transitions* that represent computations or data restructuring operations, or *places* that represent data transfers. Transitions are *ordinary* or *special*, and there are five special transitions. Places are *graph variables* or *queues*. Each node has *ports* that transmit data to and from other nodes.

We first make some assumptions about the generic parameters:

- *Indices* contains the field *category*
- *Attributes* contains *transition*, *place*, and *graph*, and the names of the PGM data types.

port-attr assigns attribute values to ports. The *dir* and *type* attributes indicate the direction and data type used by a port. The *category* attribute indicates whether the template to which the port belongs is a transition, place, or graph.

- *Parts* = *Transitions* ∪ *Places* ∪ {*Graph*}
- *Transitions* = { *Ordinary*({T_α}, ϕ, *T*), *Fanin*(*T*, *n*), *Fanout*(*T*, *n*), *Pack*(*T*),*Unpack*(*T*), *U_Merge*(*T*, *n*)}
- *Places* = {*G_Var*(*T*), *Queue*(*T*)}.

The element *Graph* of *Parts* represents a graph that is treated as a single node. An ordinary transition receives data of type {T_α}, computes the function ϕ, and outputs data of type *T*. Special transitions restructure data of type T (and possibly of size *n*). The members of *Places* store data of type *T*.

PGM_Library = ASDL_Library [Indices, Attributes, Parts] | PGM_Library_Constraints.

The first two constraints are

Primitives = *part*$^{-1}$(*Transitions* ∪ *Places*)

∀τ ∈ *Primitives*, ∀*p* ∈ interfaces(τ) • port-attr(*p*).category = part(τ)

The primitive templates in the library must be either transitions or places, and a port inherits the value of *category* from the corresponding template.

The remaining constraints describe the semantics of place templates. For each kind of place, constraints describe the interface and its semantics. A PGM graph variable holds a single rewritable data token. The relevant constraints are

part(τ) = *G_Va* in ces(τ) = {(INPUT, τ), (OUTPUT,

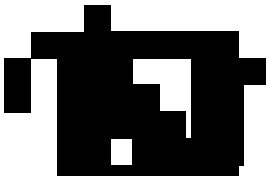
tr(INPUT, τ).dir = *in* ∧

attr(OUTPUT, τ).dir = *out* ∧

attr(INPUT, τ).type = port-attr(OUTPUT, τ = *T*

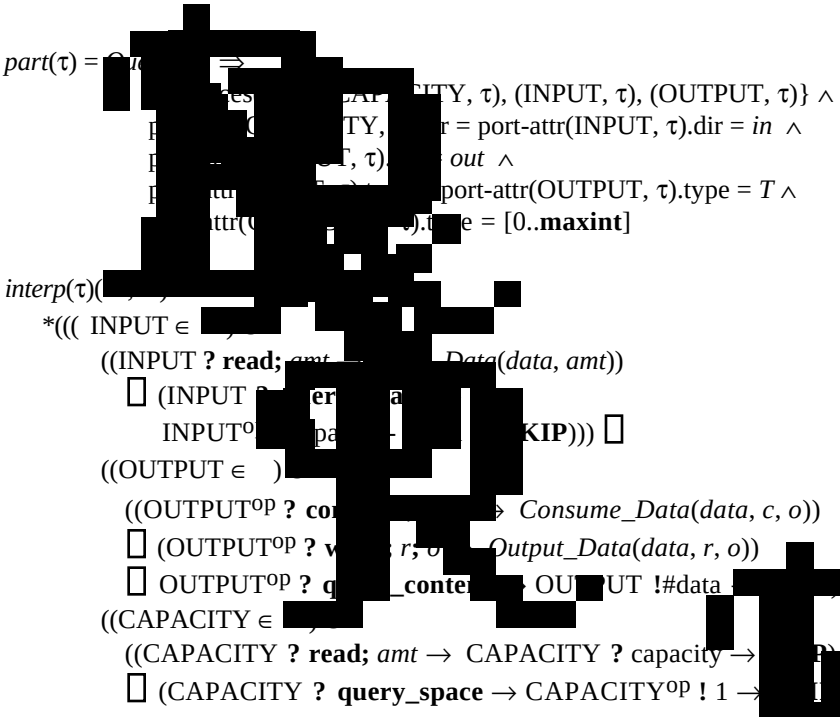
interp(τ)((□{ P data → **SKIP** : *r* ∈ **R**}) □

(□{OUTPUT$_s$ **!** data → **SKIP** : *s* ∈ }))



A gr[illegible]late has one input and one output port, each with type *T*. The t[illegible]ate's [illegible] is always ready to receive or supply data. The parameters [illegible] nels linked to the input and output ports, respectively.

The behavior of a PGM queue is more complex. Since the queue's capacity to store data tokens is limited, it must check whether sufficient space is available before receiving data. A queue template therefore has an input port, an output port, and a port that is used for communicating the queue's capacity. The relevant constraints are:



The second constraint describes a queue template's response to i[illegible] on t[illegible] INPUT, OUTPUT^{op}, or CAPACITY channels. If INPUT ∈ [illegible] queue [illegible] willing to receive data, and the process continues as described. If INPUT ∉ [illegible] queue is not willing to receive data, and the input portion of the queue process is equivalent to **SKIP**. The symbols **consume**, **read**, **write**, **query_content**,

and **query_space** are tags. If a transition port is connected to OUTPUT, the transition sends **query_content** to determine if the queue contains enough data to permit the execution of the transition. If a transition port is connected to INPUT, the transition sends **query_space** to determine if the queue has enough space to store data resulting from the transition's execution. If **read** is received on READ or CAPACITY, then the corresponding process will read the requisite amount of data. Similarly, an appropriate process responds to requests received on OUTPUT^{op} by consuming data, outputting data, or outputting the number of tokens currently stored in the queue. Descriptions of these processes are given in (PGM, 1997).

An ASDL setting describes an architecture within a specific style, and a PGM setting therefore describes a PGM graph. The PGM-specific version of the **ASDL_Setting** schema is obtained by adding new signature elements (indicated in italics) and constraints. *Name* stores the name of the graph, *link* indicates the pairs of linked PGM ports, *exec-status* indicates the graph's execution status, and *state* holds the state of its place nodes.

PGM_Setting [Indices, Attributes, Parts]

ASDL_Setting [Indices, Attributes, Parts]	
PGM_Library [Indices, Attributes, Parts]	
name	: *Char**
link	: $\mathbb{F}((Nodes \times Ports) \times (Nodes \times Ports))$
exec-status	: *(run, suspend)*
state	: $Nodes \rightarrow\!\!\!\!\!| \; \text{seq } T$
PGM_Setting_Constraints	

The constraints include:

$\forall b \in ran \text{ label} \bullet \text{semantic-descr}(b) = \textbf{bsc}$

$(\text{link} \subseteq \text{slots} \times \text{slots}) \wedge ((r, s) \in \text{link} \Leftrightarrow \text{label}(r) = \text{label}(s))$

$$(r, s) \in \text{link} \Rightarrow \begin{array}{ll} & (\text{slot-attr}(r).\text{type} = \text{slot-attr}(s).\text{type}) \\ \wedge & (\text{slot-attr}(r).\text{dir} \neq \text{slot-attr}(s).\text{dir}) \\ \wedge & (\text{slot-attr}(r).\text{category} \neq \text{slot-attr}(s).\text{category}) \end{array}$$

$(\{(r, s), (r, t)\} \subseteq \text{link}) \wedge (s \neq t) \Rightarrow \text{slot-attr}(r).\text{category} = G\_Var$

The first constraint states that communication between node ports in a setting is bidirectional and synchronous. The second and third constraints express the rules that govern edges in PGM graphs: they are modeled by slots that share the same label and the same data type, but that have opposite direction and category. The last constraint requires that a slot that is related to more than one slot by *link* must be associated with a node instantiated from a graph variable. It follows that a slot that is associated with a node instantiated from a transition or queue can only be linked to one other slot.

5. REFERENCES

Hoare, C. A. R. (1985) Communicating Sequential Processes, Prentice-Hall, Englewood Cliffs, NJ.

Kaplan, D.J. and Stevens, R.S. (1995) Processing graph method 2.0 semantics. Manuscript, US Naval Research Laboratory.

Draft Processing Graph Method Standard (1998), Naval Research Laboratory.

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

Rice, M.D. and Seidman, S.B. (1996) Using Z as a substrate for an architectural style description language, Technical Report CS-96-120, Department of Computer Science, Colorado State University.

Spivey, J.M. (1989), The Z Notation, A Reference Manual, Prentice-Hall, Englewood Cliffs, NJ.