Computer Science
Technical Report

Colorado State University

Describing the PGM Architectural Style[†]

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-96-121

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

[†] This research was partially sponsored by a grant from the Naval Research Laboratory.

Describing the PGM Architectural Style

Abstract The term *software architectural style* has recently been introduced to refer to the conventions that are used to interpret a description of a software architecture. The representation and analysis of useful architectural styles is an important problem. This paper gives an overview of a methodology developed by the authors for describing the syntax and semantics of software architectural styles with an application to the US Naval Research Laboratory's *Processing Graph Method* (*PGM*). This coarse-grain dataflow architectural style has been used for more than ten years to develop signal processing applications in government and industry.

1. Introduction

Since the late 1970s, software systems have been designed as collections of modules that interact with their environments through well-defined interfaces [DK]. This perspective gave rise to a number of *module interconnection languages* (*MIL*s) that could be used to describe the configuration of the modules and interfaces of a software system [PN]. In recent years, this syntactic view of software systems has come to be regarded as insufficient, since it says nothing about the semantics of the modules and interfaces. The modular view of software systems has been replaced by the study of abstract models of the structure of software systems, which have come to be called *software architectures* ([GP], [GS]). These models raise problems analogous to those which motivated the development of data structures, with program modules playing the role of data types. First, identifying common architectural patterns for software systems is analogous to identifying basic data structures such as lists, stacks, and trees. The importance of identifying such useful "architectural styles" is now widely recognized [AAG]. Second, identifying appropriate architectural description notations is analogous to developing high level programming languages for describing and using data structures. In our view, a notion of "abstract software types" can provide a formal basis for software architectures, just as abstract data types do for data structures. These software types provide the foundation for a system-independent language (*ASDL*) that can be used to compare architectural styles and particular architectures, and also to gain an increased understanding of complex architectural styles. This paper gives an overview of *ASDL* and shows how it can be used to obtain a description of an industrial-strength software architectural style.

Each *ASDL* software type is associated with a *Z* schema [S] that is invariant across all applications. The software types support the description of the execution and interface semantics of the components of an architecture or architectural style. In addition, *ASDL* provides schemas that correspond to basic operations for modifying the state described by the software types. The use of *Z* schemas provides the language with a flexible and modifiable type-theoretic basis for

describing, analyzing, and comparing various architectural styles. This is accomplished by specifying the values of specific variables found in the schemas and constraints on these variables, and by adding application-specific declarations, constraints, and operations to describe a particular style.

The *Z* schemas used in *ASDL* are derived from those used in the model for *MIL*s proposed by Rice and Seidman in [RS1] (henceforth called the RS *MIL* model). As in the earlier model, the *ASDL* schemas are generic *Z* schemas. The syntax of a particular architectural style is represented by a specific instantiation of the generic schemas. Such an instantiation constrains the configuration of the elements that make up an architecture. While this is similar to the approach taken with *MIL*s, this approach cannot be used to represent the semantics of a style, since *MIL*s carry no semantic information. A similar observation has recently been made by Abowd, Allan, and Garlan [AAG], who state that an abstract representation of the elements of an architecture is insufficient to convey the meaning of the architecture, and that these elements must be interpreted if they are to be meaningful. *ASDL* provides such an interpretation, since it treats syntax and semantics within the same formal framework. Structural aspects common to all styles are expressed by the generic *ASDL* schemas, while the features that are characteristic of a style are expressed by the style-specific instantiation. In this way, architectural styles can be described generically, and critical features of specific styles can be isolated and analyzed with the goal of obtaining a deeper understanding.

The utility and power of the model will be demonstrated by an application to an architectural style that has industrial significance. The *Processing Graph Method* (*PGM*) is a coarse-grain dataflow software architectural style developed at the US Naval Research Laboratory, primarily for signal processing applications [KS]. It has been used for more than ten years to design signal processing software for execution on a special-purpose multiprocessor. An effort is currently under way to develop an *IEEE* standard for *PGM*. This effort is intended to support the migration of the methodology and software to commercially available multiprocessors and to provide a common basis for application developers working on different hardware platforms.

The syntax and semantics of the *PGM* style are currently only described by natural language text [KS]. Since *PGM* is a very complex style, this text is often obscure and subject to varying interpretations. An *ASDL* description of the *PGM* style has the potential to provide a solid and reliable platform for developers and implementors.

2. *ASDL*: The Use of Software Types to Describe Architectural Styles

In this section, we will describe the software types used by *ASDL*. These types can be categorized into several groups that will be discussed in turn: *MIL* types, semantic types, unit types, and system types.

2.1 *MIL* Types

The *Z* schemas of [RS1] were used to describe module interconnection languages, which express the structure of a software system in terms of constraints imposed on the system's modules and module interfaces. Since the modules and interfaces can be regarded as representing the syntax of a software architecture, these schemas provide an initial substrate for *ASDL*.

The syntax and static semantics of the modules that make up a given software architecture are specified by the *Z* schemas **MIL_Library** and **MIL_Setting**. These schemas have been modified only slightly from the **Library** and **Setting** schemas of [RS]. The schemas use infinite sets *Labels*, *Nodes*, *Ports*, and *Templates* that are assumed to be disjoint.

The **MIL_Library** schema specifies a *library type* that provides a collection of templates and information about their interfaces. A template represents a computational component that is available for inclusion into a software architecture, and its interface consists of ports that are used for sending and receiving data. 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. 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.

MIL_Library [Indices, Attributes]

interfaces : Templates $\rightarrowtail\!\!\!\!\!\!\mid\;\;$ $\mathbb{F}_1$Ports
port-attr : Ports $\rightarrowtail\!\!\!\!\!\!\mid\;\;$ Indices $\rightarrow$ Attributes
Collection : $\mathbb{F}_1$Templates
Primitives $\subseteq$ Collection
Collection = *dom* interfaces
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 **MIL_Setting** schema specifies a *module type* based on the library type that consists of a set of *nodes* and a set of *connections* that are defined by *shared labels*. The nodes represent the components of a software architecture; each node is instantiated from a library template. A node has external interfaces called *slots* that correspond to (and inherit attributes from) the ports on the underlying template. A label shared by two or more slots creates a connection that can be used for data movement between the corresponding nodes.

MIL_Setting [Indices, Attributes]

MIL_Library [Indices, Attributes]
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
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)

2.2 Semantic Library and Module Types

In *ASDL*, the schemas representing the *MIL* types have been extended to support the specification of execution semantics. The templates in the extended **Library** schema have two additional features: *object attributes* and *semantic interpretations*. The new library schema can be thought of as a *semantic library type.*

ASDL_Library [Indices, Attributes, Parts]

MIL_Library [Indices, Attributes]
part : Templates $\rightarrow\!\!\!\!\!\!\mid\;$ Parts
interp : Templates $\rightarrow\!\!\!\!\!\!\mid\;$ Interpretations
dom interp = *dom* part = Collection

Object attributes are specified by a *part* mapping that provides application-dependent information. Semantic interpretations are assigned to the template's interface by an *interp* mapping that associates a composition of guarded *CSP* processes [H] with each template. For example, if a template τ has two ports p and q with direction attributes *in* and *out*, respectively, then the *CSP* process

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

specifies that the template provides a non-terminating copy operation. The members of *Interpretations* are described in [RS2].

The extended *Setting* schema 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.

ASDL_Setting [Indices, Attributes, Parts, SemanticDescriptions]

MIL_Setting [Indices, Attributes]
ASDL_Library [Indices, Attributes, Parts]
comp-expr : ProcessExpressions
semantic-descr : Labels $\rightarrow\!\!\!\!\!\!\mid$ SemanticDescriptions

---

dom semantic-descr = *ran* label

A composition expression is a restricted type of *timed CSP* process [RR] 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 members of *ProcessExpressions* are described in [RS2].

It is important to note that *ASDL* uses the *Z* and *CSP* formalisms orthogonally, so that there is no need to propose a common semantic domain for the two formalisms. Specifically, the use of *CSP* is confined to providing a process algebra value for the *comp-expr* variable of the **ASDL_Setting** schema and for the interpretation of each template τ (*interp(*τ)) of the **ASDL_Library** schema. The character strings assigned to these elements correspond to *CSP* process algebra expressions.

A semantic abbreviation associated with a label represents a communication protocol, as well as additional application-dependent information. The set *SemanticDescriptions* contains *abbreviations* that correspond to a variety of communication capabilities, and the mapping *semantic-descr* assigns an abbreviation to each label in a setting. For example, the abbreviations **uac** and **usc** represent unidirectional asynchronous and synchronous communication, respectively.

Each abbreviation *a* has a *meaning* $[\![a]\!]$ and a set of associated *properties*, including its text description. For example, the meaning of **usc** is described by the *CSP* expression

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

The associated properties may include an alphabet like {*in*, *out*} or an alternate specification of the meaning, such as *out* ≤ *in* (each trace on *out* is a prefix of a trace on *in*). Other properties might include timing information or a restriction on the buffer size for an asynchronous protocol.

The execution semantics of a module are 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. The *ASDL_Setting* schema can therefore be thought of as a *semantic module type* that contains the basic components and the information needed to execute the module.

2.3 Unit and System Types

The **ASDL_Setting** schema represents a module as a self-contained computational unit without any external connections. These connections and the associated interface semantics are described by an additional *unit type*. For reasons of space, we will not be able to present the corresponding schema here; full details can be found in [RS2]. This schema provides a set of *virtual ports* that represents the public interfaces of the unit, links between the virtual ports and the slots of the module, and semantic descriptions of the communication capabilities of the virtual ports.

There is also a *system type* that represents architectural state information: the modules and units that have been used to describe an architecture, the relationships between units and modules, and the connection between library templates and units. We will not be able to present the corresponding schema here; full details can be found in [RS2].

2.4 Operations

ASDL contains a number of basic operations that support the incremental specification of the software types. These serve as guides for the design of application-dependent operations that are constructed by adding new signatures and constraints to existing operations or by incorporating existing operations into a new operation. The operations include *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, *relation operations* to specify and modify relations between units, and operations that define the units needed to support a top-down design methodology. Detailed descriptions of these operations can be found in [RS2].

3. The *PGM* Architectural Style

The *Processing Graph Method* (*PGM*) is a coarse-grain dataflow software methodology developed at the US Naval Research Laboratory, primarily for signal processing applications. It has been used for more than ten years to design signal processing software for execution on a special-purpose multiprocessor. The current effort to develop an *IEEE* standard for *PGM* is intended to support the migration of the methodology and software to commercially available multiprocessors and to provide a common basis for application developers working on different hardware platforms.

A *PGM* application [KS] consists of one or more *PGM graphs* and *PGM command programs*. The nodes of a *PGM* graph 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 interfaces between transitions and places are represented by *ports*. 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 the input ports, performs the specified computation, and writes data to the output ports. Places offer several forms of data transmission between transitions: queues use a first-in, first-out communication protocol, graph variables provide rewritable data, and graph constants provide read-only data. In many circumstances, the result of the execution of a *PGM* graph can be shown to be independent of the execution order of its individual transitions ([KM], [SK]).

Figure 1 illustrates a *PGM* graph Γ. The circles represent transition nodes, and the square represents a place node corresponding to a queue. Ports are represented by small squares on node boundaries.

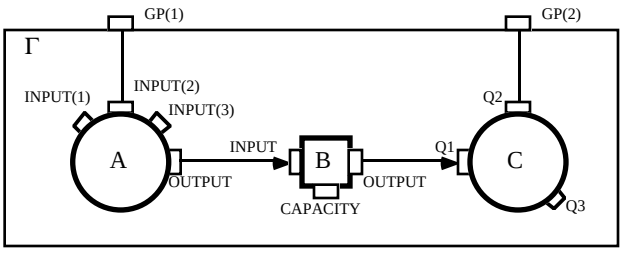


Figure 1. A *PGM* Graph

PGM command programs are programs that consist of a sequence of calls to specific procedures that create and instantiate *PGM* graphs and also provide facilities for dynamic manipulation of

PGM graphs. A command program creates a *PGM* graph by calling procedures that set up transition and place nodes, establish the graph topology by setting graph parameters and linking node ports, and then instantiate the graph as an executable object. Graph manipulation is provided by command program procedures that stop and restart executing graphs, create and delete edges between a graph's ports and its environment, and allow graph parameters to be dynamically reset.

Since the details of the computations represented by *PGM* transition nodes are not specified in a *PGM* graph, a *PGM* graph corresponds to a software architecture. From that perspective, the constraints governing the configuration of *PGM* graphs can be regarded as defining a *PGM architectural style*. A textual description of these constraints is given in [KS]. The constraints are rather complex, and the text of [KS] is often subject to conflicting interpretations. In the following section, we will show how *ASDL* can be used to obtain a reliable description of the *PGM* style that can serve as a stable foundation for implementors. We do not regard the command program procedures as forming part of the *PGM* style, and we have therefore not included them in the *ASDL* description.

4. Describing the *PGM* style in *ASDL*

As mentioned above, *PGM* nodes are either *transitions* that represent computations or data restructuring operations, or *places* that represent data transfers. Transitions are further classified as *ordinary* or *special*, and special transitions have one of five parametrized flavors, which will be described below. Places are further classified as *graph constants*, *graph variables*, or *queues*. Each node is associated with a set of *ports* which are used to transmit data to and from other nodes. Similarly, each port is associated with a set of attributes which describe its characteristics.

The first step toward obtaining an *ASDL* description of *PGM* is to define a *PGM*-specific version of the semantic library schema. To do this, we must first make some *PGM*-specific assumptions about the generic parameters that must be supplied for this schema:

- *Indices* contains the field *category*
- *Attributes* contains the members of the set *Parts* and the names of the *PGM* types
- $Parts = Transitions \cup Places \cup \{Graph\}$

The function *port-attr* in the **MIL_Library** schema assigns attribute values to ports. According to this schema, $dir \in Indices$ and $\{in, out\} \subseteq Attributes$. The *dir* attribute governs whether a port inputs or outputs data. The *type* attribute indicates the type of the data that may be communicated

using the port. The *category* attribute identifies the part corresponding to the template to which the port belongs.

- *Transitions* = {*Ordinary*($\{T_\alpha\}_{\alpha \in \Lambda}$, $\phi : F \rightarrow T$), *Fanin*(*T*, *n*), *Fanout*(*T*, *n*), *Pack*(*T*), *Unpack*(*T*), *U_Merge*(*T*, *n*)}
- *Places* = {*G_Var*(*T*), *G_Const*(*T*), *Queue*(*T*)}

The element *Graph* of *Parts* corresponds to a subgraph that is treated as a single node. The definitions of *Transitions* and *Places* need some further comment. With respect to *Transitions*, the $\{T_\alpha\}$ are the types of the data tokens communicated through the ports of an ordinary transition, and the computation performed by that transition corresponds to the function ϕ, whose set of input arguments has the finite set type *F*. The remaining members of *Transitions* represent the five flavors of the special transitions used for data restructuring in *PGM*; each is parametrized by a type *T*, and three are also parametrized by an integer *n*. The parametrized definitions of the members of *Places* express the type of the data that can be stored in a place.

We can then define **PGM_Library** by adding some *PGM*-specific constraints to the parameterized generic schema:

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

The first of these constraints is

$$Primitives = part^{-1}(Transitions \cup Places),$$

which states that the primitive templates in the library must be either transitions or places.

The second constraint is

$$\forall \tau \in Primitives, \forall p \in \text{interfaces}(\tau) \bullet \text{port-attr}(p).\text{category} = \text{part}(\tau),$$

which expresses the fact that the *category* attribute of a port is inherited from the corresponding template. This will be needed in **PGM_Setting** to express the fact that the *PGM* style requires linked nodes to have opposite category (see S4b below).

The remaining constraints describe the semantics of primitive templates. A pair of constraints is needed for each primitive template. The first of these constraints describes the interface, while the second specifies a *CSP* process that describes the semantics of the template. The template's interface is modeled by using either a single *CSP* channel or a family of channels. The names of these channels are derived from the names of the ports comprising the interface. For example, a

two-member set of channels corresponding to the port OUTPUT is denoted as {OUTPUT, OUTPUT^{op}}.

The two constraints associated with the part *G_Const*(*T*) are

(GC1) *part*(τ) = *G_Const*(*T*) ⇒ interfaces(τ) = {(OUTPUT, [redacted]} ∧ p[redacted](OUTPUT, τ).dir = *out* ∧
port-attr(OUTPUT, τ).ty[redacted] = *T*

(GC2) *interp*(τ)(S) = *([redacted]{OUTPUT_S **!** data → **SKIP** : *s* ∈ [redacted]},

(GC1) states that a *G_Const*(*T*) (graph constant) template has a single port with dire[redacted]*ut* and type *T*. (GC2) states that a template is always ready to supply its data value. Recall that *SP*, [redacted] represents nondeterministic choice, * represents indefinite iteration, **;** represents sequential composition, and that **?** and **!** are used to denote channel input and output. [redacted] to the set of channels linked to the port. These constraints express the idea that a *PGM* graph constant holds a single unchangeable data token (determined when the graph is instantiated) that will be supplied when requested.

While a *PGM graph[redacted]* holds a single data token which [redacted] requested, the value of this tok[redacted] change[redacted] he GV constraints tha[redacted] exp[redacted] ics should be contrasted with their [redacted] gs.

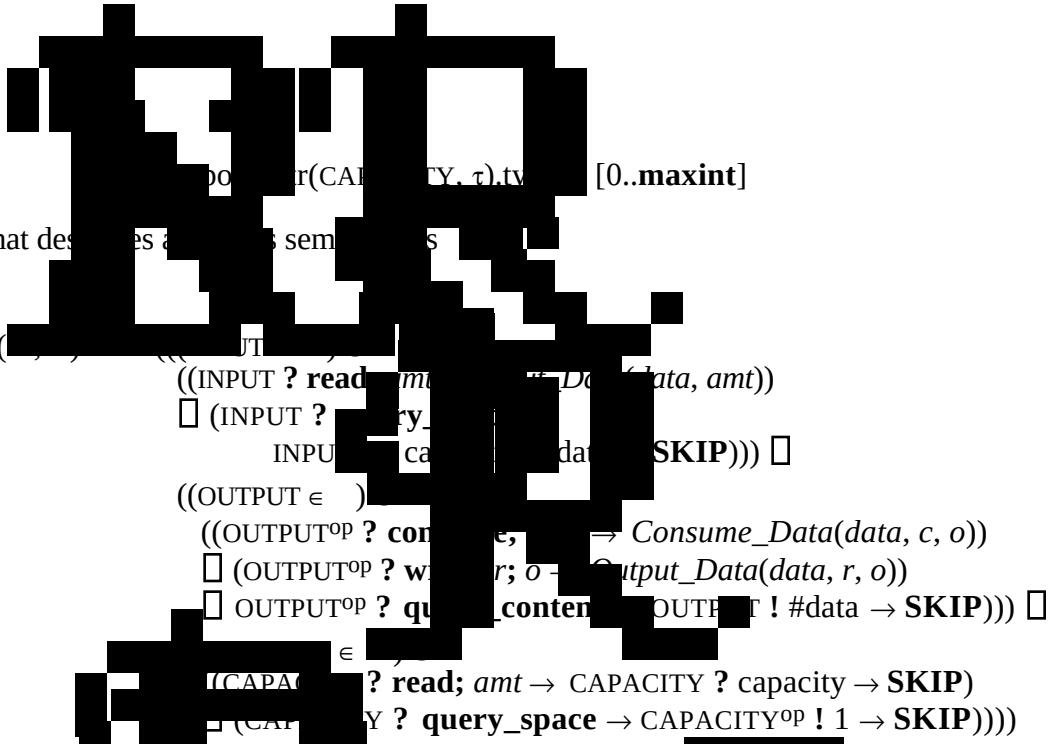
(GV1) *part*(τ) = *G_V[redacted]* [redacted]faces(τ) = {(INPUT, τ), (OUTP[redacted]
[redacted]ttr(INPUT, τ).dir = *in* ∧ port[redacted] τ).dir = *out* ∧
[redacted]ttr(INPUT, τ).type = port-att[redacted]pe = *T*

(GV2) *interp*(τ)([redacted] [redacted] [redacted]ta → **SKIP** : *r* ∈ [redacted]},
[redacted]PUT_S **!** data → **SKIP** : *s* ∈ [redacted]},

(GV1) states that a G[redacted]*T*), [redacted]riable) template has one input and one output port, each with type *T*. (GV2) s[redacted]hat the [redacted]ate's interface is always ready to receive or supply data. The parameters [redacted] [redacted]nels linked to the input and output ports, respectively.

The behavior of a *PGM queue* is more complex. Since its capacity to store data tokens is limited, it is necessary to check whether sufficient space is available before receiving data. As a consequence, a queue template has an input port, an output port, and a port that is used for communicating the queue's capacity. The constraint that describes this interface is:

(Q1) *part*(τ) = *Queue*(*T*) ⇒ interfaces(τ) = {(CAPACITY, τ), (INPUT, τ), (OUTPUT, τ)} ∧
port-attr(CAPACITY, τ).dir = port-attr(INPUT, τ).dir = *in* ∧
port-attr(OUTPUT, τ).dir = *out* ∧
port-attr(INPUT, τ).type = port-attr(OUTPUT, τ).type = *T* ∧



The constraint that describes a

(Q2) *interp*(τ)(

((INPUT **? read**

(INPUT **?**

INPU

((OUTPUT ∈

((OUTPUT^op **? con**

(OUTPUT^op **? w**

OUTPUT^op **? qu**

∈

((CAPA

(CAP

This constraint describes a response to input on INPUT, OUTPUT^op, or CAPACITY channels. The

act as preconditions for the

el operations. The syntax for the guards is an e

that w

adapted from the occa

uage [J]. For example, if INPUT ∈

eue is

eceive data, and

ontinues as described. If INPUT ∉

ever, the que

ot willing to receiv

ta, and

nput portion of the queue process is equivalent to SKIP. Similarly, if OUTPUT ∈

ue can use the OUTPUT port to send data (resp. receive data) on the channel OUTPUT (resp. OUTPUT^op).

The symbols **consume**, **read**, **write**, **query_content**, and **query_space** act as tags. If a transition port is connected to the OUTPUT port, then the transition sends the tag **query_content** to determine if the queue contains enough data to permit the execution of the transition. If a transition port is connected to the INPUT port, then the transition sends the tag **query_space** to determine if the queue has enough space to store data resulting from the transition's execution. If a **read** request 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].

Similar constraints are associated with ordinary transitions. These nodes represent computations, and they may have any number of input and output ports of different types. For reasons of space, neither these constraints, nor those describing the semantics of special transitions, can be given here. Full details can be found in [RS2].

The *PGM*-specific version of the *ASDL* semantic module schema is obtained by adding five new signature elements (indicated in boldface italics) and five new constraints to the **ADSL_Setting** schema. The **PGM_Setting** schema corresponds to a *PGM* graph. The kind of the setting is stored in *setting-type*, and the name of the graph is stored in the character string *name*. The *link* relation denotes the pairs of slots corresponding to pairs of *PGM* ports that are joined by *PGM* graph edges. Since *PGM* command programs can suspend and restart the execution of a graph, a setting must contain information about the execution status of a graph (*exec-status*) and the current state of the graph's place nodes. State is represented by the mapping *state* : $Nodes \rightarrow\!\!\!\!\!\!\mid\ D_\perp$, where $\boldsymbol{D}_\perp$ denotes the domain of possible data values, including the undefined value $\perp$.

PGM_Setting [Indices, Attributes, Parts, SemanticDescriptions]

ASDL_Setting [Indices, Attributes, Parts, SemanticDescriptions] **PGM_Library [Indices, Attributes, Parts]** ***setting-type*** **: *Parts*** ***name*** **: *Char**** ***link*** **: $\mathbb{F}$ *((Nodes* $\times$ *Ports)* $\times$ *(Nodes* $\times$ *Ports))*** ***exec-status*** **: *(run, suspend)*** ***state*** **: *Nodes* $\rightarrow\!\!\!\!\!\!\mid$ $\boldsymbol{D}_\perp$**
*PGM*_Setting_Constraints

*PGM*_Setting_Constraints contains the following specific constraints:

(S1) setting-type = *Graph*

(S2) $b \in ran$ label $\Rightarrow$ $|$ label$^{-1}(b)$ $| \geq 2$

(S3) $\forall b \in ran$ label •
semantic-descr(b) $\in$ {**usc**, **bsc**} $\wedge$
semantic-descr(b) = **usc** $\Leftrightarrow$ $\exists\, s \in$ label $^{-1}(b)$ • slot-attr(s).category = *G_Const*

(S4) (a) (link $\subseteq$ slots $\times$ slots) $\wedge$ ((r, s) $\in$ link $\Leftrightarrow$ label(r) = label(s))

(b) (r, s) $\in$ link $\Rightarrow$ (slot-attr(r).type = slot-attr(s).type) $\wedge$ (slot-attr(r).dir $\neq$ slot-attr(s).dir)
$\wedge$ (slot-attr(r).category $\neq$ slot-attr(s).category)

(c) ({(r, s), (r, t)} $\subseteq$ link) $\wedge$ ($s \neq t$) $\Rightarrow$ slot-attr(r).category $\in$ {*G_Const*, *G_Var*}

(S5) (a) *dom* state = {$n \in$ *dom* node-parent : part(node-parent(n)) $\in$ *Places*}

(b) $\forall n \in$ *dom* state •
(part(node-parent(n)) $\in$ {*G_Var*(T), *G_Const*(T)} $\Rightarrow$ *state*(n) has type $T \times \mathbb{N}$) $\wedge$
(part(node-parent(n)) = *Queue*(T) $\Rightarrow$ *state*(n) has type **seq**$T \times \mathbb{N}$)

(S1) expresses the fact that a setting can only be used to construct a *PGM* graph. (S2) requires that each label be shared by at least two slots. This technical constraint is needed to support the specification of PGM command program procedures. (S3) states that communication between node ports in a setting is synchronous, and is bidirectional (**bsc**) unless one of the nodes is a graph constant. In the latter case, the communication is unidirectional (**usc**).

The link constraints in (S4) express the rules that govern edges in *PGM* graphs. (S4a) asserts that the pairs related by *link* must be slots, and that slots related by *link* must share the same label. (S4b) requires that slots related by *link* must share the same data type, but must have opposite direction and category. (S4c) requires that a slot that is related to more than one slot by *link* must be associated with a node instantiated from a graph constant or 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.

The constraints in (S5) describe the way in which the state of a setting is modeled. A dynamic representation of the state of a *PGM* graph is needed to support the modeling of runtime graph management procedures. Since *PGM* transition nodes do not have state, (S5a) makes the domain of the *state* mapping equal to the set of nodes instantiated from *PGM* place templates. (S5b) gives type descriptions for the state of graph constants, graph variables, and queues.

5. Benefits of the *ASDL* approach

The goal of the effort to develop an *IEEE* standard for *PGM* is a document that will be accepted as a definition of *PGM* by two communities: the engineers who are building *PGM* architectures that correspond to signal processing applications, and the manufacturers of multiprocessor hardware platforms that will execute these architectures. This document must include a correct and unequivocal definition of *PGM*. Until now, such definitions have been written in natural language [KS], and the informality of natural language definitions has given rise to many rounds of discussion and clarification. The primary purpose of constructing an *ASDL* model of *PGM* was to create a formal model that could serve as a reference point for developers and implementors.

As an example, compare the following definition of the *PGM* queue, taken from [KS], with the formalism of constraints (Q1) and (Q2) of **PGM_Library**.

> A *queue* is a place that shall execute by receiving tokens from its data input port and sending them out through its data output port. Tokens shall be handled in a first-in, first-out manner and shall be stored in the queue's associated family. A queue port shall be connected to at

most one transition port. Each queue shall be associated with a family of tokens. The number of tokens currently stored in this family shall be called the queue's *content.* The *capacity* of a queue shall initially be set at MAXINT, which denotes the maximum integer value that can be represented by the target hardware. Each queue shall have an input capacity port of integer mode. If the name of a queue is *queuename*, its capacity port shall be named *queuename*.CAPACITY. At any time after initialization, a queue's capacity shall be equal to the value of the most recent integer token written to the queue's capacity port. The result of using the *consume* operation to remove *N* tokens from a queue shall be the deletion of the first *N* tokens from the queue.

It is far easier to base an architecture or an implementation on the formal constraints than on an interpretation of the words in the natural language definition. We have found that discussions based on the formal constraints have been far briefer and more insightful than discussions based on the natural language definitions. As a consequence, we will be including a version of the formalism as an annex to the draft *PGM* standard.

6. Related research

The term *architectural style* was introduced by Abowd, Allen, and Garlan in [AAG]. In that paper, the authors formalize an abstract syntax (henceforth referred to as the AAG syntax) for software architectures and show how the abstract syntax can be mapped to a semantic model for each style. The abstract syntax can be compared with the architectural syntax provided by the RS *MIL* model, since both provide a generic framework for treating architectures. The AAG syntax consists of *Z* schemas that specify sets of components and connectors, and an attachment mapping that is used to associate elements of those sets. The semantic models in [AAG] are also expressed using *Z* schemas, and the mapping from syntax to semantics is defined using the signature elements of the syntactic and semantic schemas. Finally, the constraints imposed on the syntax by the semantic model are derived by combining the syntactic and semantic schemas. Despite their common goals and their common use of *Z* formalism, there are some important differences between the approaches. First, the approach used in [AAG] often makes it difficult to separate the syntactic and semantic aspects of a style. For example, syntactic features of styles are usually discussed only after the syntactic and semantic schemas have been combined In our approach, such syntactic features can be treated independently of semantics by adding style-specific signature elements and constraints to a RS *MIL* library or setting schema. In addition, semantic models in [AAG] are built around transition functions representing state machines. This approach seems less expressive than the incorporation of process algebra expressions into *ASDL* schemas, as advocated here.

In [AG], Allen and Garlan propose another approach for describing architectural components that is also somewhat similar to *ASDL*. This approach is associated with a description language called

WRIGHT. The *instances* and *attachments* described in [AG] correspond to *ASDL* nodes and labels. Also, the idea of a *connector* provides a means of describing execution semantics. The ports on instances correspond to port processes and although this phrase is not used, roles correspond to "ports on connectors". Connections are made between ports on nodes and roles on connectors, but two nodes are never directly connected. In addition, *CSP* expressions are used to describe the semantics of the ports and connectors. This is roughly equivalent to our description of semantics using semantic descriptions and a composition expression. By contrast with *ASDL*, [AG] has no notion of encapsulation, and there is no type-theoretic basis that supports the specification of generic operations for constructing an architecture. Nonetheless, the spirit of the work in [AG] is similar to our own, most importantly in its recognition of the importance of using a formal notation to obtain a precise description of execution and interface semantics.

The primary purpose for constructing an *ASDL* model of *PGM* was to obtain a deeper understanding of an industrially important architectural style. A similar approach was taken by Delisle and Garlan when they constructed a *Z*-based model of the architecture of a digital oscilloscope [DG]. More recently, Allen [A] has undertaken a similar effort by using *WRIGHT* to model the Department of Defense's *High Level Architecture for Simulations* (*HLA*) as an architectural style in order to obtain an understanding of the architectures built using this style.

7. Conclusions

This paper proposes a formal and generic approach to describing the syntax and semantics of software architectural styles. The approach was illustrated by applying it to the *PGM* architectural style. The process of developing the *PGM* application is also of interest. We began with a textual description of the *PGM* style, and we used that description to develop the *PGM*-specific *ASDL* schemas. Since these schemas are formal representations, we can use them to mathematically verify the consistency and correctness of various aspects of the *PGM* model. On the other hand, the schemas also provide the basis for a custom notation that can be used to convince *PGM* practitioners of the correctness and power of our model, and the resulting descriptions will form part of the draft *PGM* standard. We feel that one of the potential advantages of our formalism is that it enables the modeler to acquire a deep understanding of a software architectural style. It can therefore serve as an effective mediator between textual descriptions of architectures and customized notations.

References

[A] R. Allen, "HLA: A standards effort as architectural style", *Proceedings of the Second International Software Architecture,* pp. 130-133, 1996.

[AAG] G. Abowd, R. Allan, and D. Garlan, "Using style to understand descriptions of software architecture", *Proceedings of the ACM SIGSOFT'93 Symposium on Foundations of Software Engineering*, pp. 9-20, 1993.

[AG] R. Allan and D. Garlan, "Formalizing architectural connection", *Proceedings of the 16th International Conference on Software Engineering*, pp. 71-80, 1994.

[DG] N. Delisle and D. Garlan, "Applying formal specification to industrial problems: a specification of an oscilloscope", *IEEE Software* **7**(1990), 29-37.

[DK] F. DeRemer and H. Kron, "Programming-in-the-large versus programming in-the-small", *IEEE Transactions on Software Engineering* **2**(1976), 80-86.

[GP] D. Garlan and D. E. Perry, "Introduction to the special issue on software architecture", IEEE Transactions on Software Engineering 21 (1995), 269-274.

[GS] D. Garlan and M. Shaw, "An introduction to software architecture", in V. Ambriola and G. Tortora, eds., *Advances in Software Engineering and Knowledge Engineering*, vol. 1, World Scientific, pp. 1-39, 1993.

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

[J] G. Jones, *Programming in occam*, Prentice-Hall, 1987.

[KM] R. M. Karp and R. E. Miller, "Properties of a model for parallel computation: determinacy, termination, queuing", *SIAM Journal of Applied Mathematics* **14** (1966), 1390-1411.

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

[PGM] Draft Processing Graph Method Standard, Naval Research Laboratory, January 1997.

[PN] R. Prieto-Diaz and J. M. Neighbors, "Module interconnection languages", *Journal of Systems and Software* **6**(1986), 307-334.

[RR] G. M. Reed and A. W. Roscoe, "A timed model for communicating sequential processes", *Theoretical Computer Science* **58** (1988), 249-261.

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

[RS2] M.D. Rice and S.B. Seidman, "Using Z as a substrate for an architectural style description language", Technical Report 96-120, Department of Computer Science, Colorado State University, 1996.

[S] J. M. Spivey, *The Z Notation, A Reference Manual*, Prentice-Hall, 1989.

[SK] R. S. Stevens and D. J. Kaplan, "Determinacy of generalized schema", *IEEE Transactions on Computers* **41** (1992), 776-779.