Computer Science
Technical Report

Colorado
State
University

Using Z as a Substrate for an Architectural Style Description Language[1]

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-96-120

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

[1] The research reported in this paper was partially supported by a grant from the U. S. Naval Research Laboratory.

Using Z as a Substrate for an Architectural Style Description Language[2]

Michael D. Rice Stephen B. Seidman

Abstract: This paper shows how *Z* can be used as a substrate for an architectural style description language. The language provides a collection of abstract software types that support the description of execution and interface semantics, logical views, and relationships between logical views. The software types correspond to application-invariant *Z* schemas, which provide a type-theoretic basis for the language that allows it to be used for describing, analyzing, and comparing various architectural styles and logical views.

1.. Introduction

The field of software architectures deals with the design, construction, maintenance, and structure of large software systems. The study of these systems raises 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 useful "architectural styles" is now widely recognized [1]. Second, identifying appropriate architectural description notations is analogous to developing high level programming languages for describing and using data structures. This is an important issue that has received far less attention. Most research efforts have developed software systems for designing domain-specific architectures, as opposed to system-independent architectural notations. 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 serve as the foundation for a system-independent language that can be used to compare architectural styles and particular architectures.

This paper presents an overview of an architectural style description language (*ASDL*) based on a collection of software types. Modeling software architectural styles may requre a variety of notations and methodologies, and it is important that the underlying formalism has enough expressive power.. We have done this by associating each software type with a *Z* schema [12] that is invariant across all applications. Collectively, the software types permit the description of *execution* and *interface semantics*, *logical views*, and *relationships* between logical views. In addition, there are 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 and logical views[3]. This is accomplished (i) by specifying the values of specific variables found in the schemas and constraints on these variables, and (ii) by adding application-specific declarations, constraints, and operations to describe a particular logical view.

2. Language Overview

This section provides an overview of the *Z* schemas that correspond to the software types and operations of *ASDL*. The dependencies between the schemas are shown in Appendix A.

---

[2] A *logical view* is a description of one characteristic of an architecture (such as a *functional* view of a compiler based on lexical analysis and parsing components or a *process* view of an operating system based on processor states like running or suspended). The identification of logical views is a precursor to the formulation of an architectural style.

Examples are given to show how the *ASDL* schemas can be used to specify (a) execution semantics, (b) dynamic views, (c) interface semantics, and (d) relationships between views.

MIL Types

In [10], *Z* schemas were used 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. Since the modules and interfaces can be regarded as representing the syntax of a software architecture, the *Z* schemas of [10] 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 [10]. 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, 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	:	$\text{Templates} \rightarrowtail\!\!\!\!\!\!\!+\ \mathbb{F}_1\text{Ports}$
port-attr	:	$\text{Ports} \,-\!\!\!\!\!+\!\!\!\!\rightarrow\, \text{Indices} \rightarrow \text{Attributes}$
Collection	:	$\mathbb{F}_1\text{Templates}$

$\text{Primitives} \subseteq \text{Collection}$
$\text{Collection} = \mathit{dom}\ \text{interfaces}$
$\mathit{disjoint}\ \ \mathit{ran}\ \text{interfaces}$
$\mathit{dom}\ \text{port-attr} = \bigcup\ \mathit{ran}\ \text{interfaces}$
$\forall p \in \mathit{dom}\ \text{port-attr} \bullet \text{port-attr}(p).\mathit{dir} \in \{\mathit{in}, \mathit{out}\}$

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*. 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.

MIL_Setting [Indices, Attributes]

MIL_Library [Indices, Attributes]
node-parent : Nodes $\to\!\!\!\!\!\!\mid\;$ Templates
slots : $\mathbb{F}$(Nodes × Ports)
slot-attr : Nodes × Ports $\to\!\!\!\!\!\!\mid\;$ Indices → Attributes
label : Nodes × Ports $\to\!\!\!\!\!\!\mid\;$ Labels
slots = *dom* slot-attr
dom label ⊆ slots
∀*n* ∈ *dom* node-parent •
node-parent ∈ Collection ∧ *p* ∈ interfaces(node-parent(*n*))
⇒ (*n*, *p*) ∈ *dom* slot-attr ∧ slot-attr(*n*, *p*) = port-attr(*p*)

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*. 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]

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

The object attributes are specified by a *part* mapping that provides application-dependent information. The semantic interpretations are assigned to the template's interface by an *interp* mapping that associates a composition of guarded *CSP* processes [4] 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 Appendix B.

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. A node instantiated from a reference template is called a *pseudonode*.

ASDL_Setting [Indices, Attributes, Parts, SemanticDescriptions]		
MIL_Setting [Indices, Attributes]		
ASDL_Library [Indices, Attributes, Parts]		
comp-expr	:	ProcessExpressions
semantic-descr	:	Labels $\mapsto$ SemanticDescriptions
dom semantic-descr = *ran* label		

A composition expression is a restricted type of *timed CSP* process [9] 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 Appendix B.

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 our goal is to have *Z* and *CSP* reinforce each other, there is no need to propose a common semantic domain for the two formalisms.

A semantic abbreviation associated with a label represents a communication protocol, as well as additional application-dependent information involving data transfer rates and timing requirements. To do this, the set *SemanticDescriptions* contains *abbreviations* that correspond to a variety of communication capabilities. The mapping *semantic-descr* assigns an abbreviation to each label in a setting. The following abbreviations illustrate some of the possible semantic descriptions:

uac (**usc**)	--	unidirectional asynchronous (synchronous) communication
abp	--	alternating bit protocol
rpc	--	remote procedure call
brod	--	broadcast input data
merge	--	combine input data
mult	--	multiplex input data

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

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

The associated properties may include an alphabet like $\{in, out\}$ or an alternate specification of the meaning, such as $out \leq in$ (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.

In some cases, the meaning of an abbreviation is parameterized by a potential set of connections. For example, the meanings of the broadcast and multiplex abbreviations are specified by:

$$[\![\mathbf{brod}]\!](S) = *(in \;\mathbf{?}\; x \rightarrow \| \, (out_s \;\mathbf{!}\; x \rightarrow \mathbf{SKIP} : s \in S))$$
$$[\![\mathbf{mult}]\!](S) = *(0\, in_s \;\mathbf{?}\; x \rightarrow out \;\mathbf{!}\; x \rightarrow \mathbf{SKIP} : s \in S)).$$

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.

The following examples illustrate the use of library and module types.

(a) A module type in the *Processing Graph Method* (*PGM*, [7], [11]) corresponds to a graph whose nodes perform signal processing operations. The primitive templates underlying the nodes belong to two categories: *transitions* that represent computations and data restructuring operations and *places* that represent data transfers between transitions. Slots of the [illegible] instantiated from templates [illegible] categories may be connected by di[illegible] that [illegible]cify data movement.

For exa[illegible] = $G_Var(T)$ specifies th[illegible] τ is [illegible]graph variable [illegible] (correspo[illegible] *PGM* graph node holding re[illegible]stant data) with one i[illegible] (INPUT) [illegible] out[illegible] OUTPUT) port, each of type [illegible] and

$$interp(\tau)(\ldots) [\text{illegible}] x \to \textbf{SKIP} : r \in [\text{illegible}] (\{\text{OUTP}[\text{illegible}]\} x \to \textbf{SKIP} : s [\text{illegible}])$$

specifies that τ is willing to receive or send data on the indicated channels, where [illegible] correspond to the nodes that are linked to the INPUT and OUTPUT ports, respectively.

The interpretation of a *PGM* transition template enforces a dataflow execution methodology, in which the execution of a node is triggered by the arrival of sufficient data at its input slots. The composition expression specifies the parallel execution of the nodes in a module and the semantic description of each label specifies a unidirectional synchronous transfer of data.

(b) A composition expression can also describe the dynamic evolution of a module which is the target of *ASDL* operations . In this case, the alphabet of the process representing the expression includes *special event names* corresponding to the *ASDL* operations used. For example, the following portion of a composition expression describes the creation of a client node C based on a template τ and its connection to one of the server nodes S_1 and S_2 using a label L with the semantic abbreviation **bac** (bidirectional asynchronous communication):

$$create_node(C, \tau) \;\to\; (assign_label(L, S_1, C, \textbf{bac}) \sqcap assign_label(L, S_2, C, \textbf{bac}))$$

The *ASDL create_node* operation instantiates a node from a library template, and the *assign_label* operation establishes a connection between the client and server nodes by setting the values of the slot labels. These operations will be discussed further below.

Unit Type

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.

ASDL_Unit [Indices, Attributes, Parts, SemanticDescriptions]

ASDL_Setting [Indices, Attributes, Parts, SemanticDescriptions]
ASDL_Boundary [Indices, Attributes]
connect : Nodes $\times$ Ports $\rightarrow\!\!\!\!\!+$ $\mathbb{F}$Ports
virtual-port-descr : Ports $\rightarrow\!\!\!\!\!+$ 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)}

ASDL_Boundary [Indices, Attributes]

interface-attr : Ports $\rightarrow\!\!\!\!\!+$ Indices $\rightarrow$ Attributes
virtual-ports : $\mathbb{F}$Ports

virtual-ports = *dom* interface-attr

The schema *ASDL_Unit* imposes only a minimal restriction on the interface that enforces consistency with respect to the direction of data movement. Further restrictions are based on application-dependent information about the desired behavior of units. For example, type-consistency requirements may be placed on the *connect* mapping, and the *virtual-port-descr* mapping may specify broadcasting or multiplexing behavior for a virtual port.

The following example illustrates the use of unit types.

(c) A computer in a network can be represented as a unit type, where the nodes correspond to sockets, and their connections to virtual ports correspond to the assignment of sockets to services such as ftp and telnet. Since a port can be linked to more than one socket, ports must support the multiplexing of messages. The semantics of a virtual port p is specified as

$$\textit{virtual-port-descr}(p) = \textit{Receive} \;\Box\; \textit{Send}.$$

where

$$\textit{Receive} = *(\Box\; p_k \;\mathbf{?}\; \textit{msg} \rightarrow \text{socket-slot}_{\text{number}(\textit{msg})} \;\mathbf{!}\; \textit{msg} \rightarrow \mathbf{SKIP} : k \in K),$$

number(*msg*) is the local socket number identifying the destination of the message *msg*, and K corresponds to the computer's set of network connections.

System Type

The *ASDL_System* schema includes the *ASDL_Library* schema, as well as architectural state information: the modules and units that have been used to describe logical views of an architecture, the relationships between units and modules, the connection between library templates and unit types, and architectural connections between different logical views.

ASDL_System [Indices, Attributes, Parts, SemanticDescriptions]

ASDL_Library [I, A, P, S]
Units : $\mathbb{F}$**ASDL_Unit** [I, A, P, S]
basis : Templates ⇸ **ASDL_Unit** [I, A, P, S]
relation : Labels ⇸ [illegible]es × [illegible]

Collection \ *dom* basis = [illegible]
Units = *ran* basis
∀ τ ∈ Collection \ Primitiv[illegible] [illegible]is(τ).virtual-ports ∧
[illegible]s(τ) = basis(τ).interface-[illegible]

∀ ρ ∈ *dom* relation ∃ { , [illegible] } [illegible]
(*dom* relation(ρ) ⊆ *dom* [illegible].node-parent [illegible] *ran* relation(ρ) ⊆ *dom* [illegible]*.node-parent[illegible])

where I, A, P, S refer to *Indices*, *Attributes*, *Parts*, and *SemanticDescriptions*.

The 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. The mapping *relation* summarizes the relations that have been specified between sets of nodes in various units to provide architectural connections between different logical views.

The use of the resulting *system type* is illustrated by the following example:

(d) In an assembler, a module type may specify a *phase view*, where each node represents a distinct phase (first pass, second pass, ...) and connections between nodes represent control information. In this view, the composition expression specifies sequential execution and the semantic description specifies "transfer of control". Another module type may specify a *structural view*, where the nodes correspond to data structures and connections between nodes represent pointer references. In this case, the composition expression is SKIP, denoting the absence of an execution context, and the semantic description is simply "reference to". There are natural "modify" and "use" relations between the two views - if the execution of a phase node n_p alters (respectively reads) the contents of a structure node n_s, then (n_p , n_s) belongs to the "modify" (respectively "use") relation.

Operations

ASDL contains a number of basic operations that support the incremental specification of the software types by updating the *ASDL_System* schema. 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 basic operations include *setting operations* to create and delete nodes and pseudonodes, 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.

The *create_node* operation is a typical setting operation, since it alters an individual unit, but does not change the semantic library type. The availability of schema composition in *Z* makes it possible to localize the definition of these operations. In order to do so, the auxiliary

schem[illegible] le[illegible] nit [illegible] at [illegible] st [illegible] e [illegible] rm[illegible] r the *ASDL* operation is then

AS[illegible]**L(** [illegible] **)** [illegible] **ct**[illegible]**it;** [illegible] **te_System**,

where [illegible] ocal” schema that [illegible] rf[illegible] operation for a specific unit. For example, the local schema corresponding to [illegible]*_node* is

Create_Node

Ξ**ASDL_System**
Δ**ASDL_Unit**
τ**?** : Templates
τ**?** ∈ Collection
∃ *n* ∈ Nodes \ *dom* node-parent •
node-parent' = node-parent ⊕ {*n* |→ τ?}

The system-level schemas are **Select_Unit**, which returns the unit associated with a reference template **r?**, and **Update_System**, which updates the unit associated with a reference template **r?**.

Select_Unit

Ξ**ASDL_System**
r? : Templates
ASDL_Unit'
r? ∈ *dom* basis \ Collection
node-parent' = basis(**r?**).node-parent
. . .
virtual-port-descr' = basis(**r?**).virtual-port-descr

Update_System

Δ**ASDL_System**
r? : Templates
ASDL_Unit'
r? ∈ *dom* basis \ Collectio[illegible]
basis' = basis ⊕ {**r?** |→ [illegible] } • [illegible] . [illegible] p[illegible] e [illegible] de-[illegible] rent
. .
.vi[illegible] port-[illegible] = virtual-port-descr

In these schemas, the elision indicates that an analogous constraint holds for each of the ten **ASDL_Unit** variables not listed.

The encapsulation operation (**ASDL_External**) creates a new library template from an existing unit type. The virtual ports of the unit type become the ports of the template and the attributes of these ports are derived from the unit’s interface. The template's interpretation is derived from the interpretations of the templates underlying the nodes, the composition expression, the abbreviations of labels, and the semantics of the virtual ports. This represents

a complex synthesis of the semantics of the entities associated with the unit. In general, it may not be possible to construct a “closed form” interpretation due to the presence of partial and heterogeneous information.

ASDL_External

∆ASDL_System **r?** : Templates kind? : Parts
r? ∈ *dom* basis \ Collection interfaces' = interfaces ⊕ {**r?** |→ basis(**r?**).virtual-ports} port-attr' = port-attr ⊕ basis(**r?**).interface-attr part' = part ⊕ {**r?** |→ kind?} interp' = interp ⊕ {**r?** |→ **Synthesis**(basis(**r?**))}

Note that the constraints in **ADSL_External** guarantee that Collection' = Collection ∪ {**r?**}. One of the benefits of using *Z* as a basis for *ASDL* is that the formalism makes it possible to draw such conclusions.

The new library template can, in turn, be used to create a node in another module. *ASDL* permits an application-dependent interpretation of the extent to which the internal structure of the node is visible in the new module. For example, if encapsulation requires that each virtual port is linked to a node in the underlying unit, then one interpretation is that only the resources of the nodes linked to the port can be accessed through the port. On the other hand, if encapsulation permits a virtual port with no links, then another interpretation may allow characteristics of the node to be modified by using the port.

In [10], a top-down design methodology was modeled by using a family of generic connector templates. *ASDL* supports a more straightforward representation of this methodology, since it provides an operation to incorporate a special unit type into an existing module. The included unit corresponds to a module that may be empty. In the latter case, a reference template is used to instantiate a pseudonode that can be linked to other nodes. The internal structure of the unit can be constructed later, thereby providing the desired top-down design capability.

Specifically, the **Create_Unit** operation is used to create an empty unit and associate it with a reference template, the **Update_Boundary** (local) operation is used to define the unit’s interface, and the **Create_Pseudonode** (local) operation is used to use the reference template to instantiate a pseudonode as[illegible]ted [illegible]unit.

Create_Unit

∆ASDL_System **r!** : Templates
r! ∉ *dom* basis \ Collectio[illegible] basis' = basis ⊕ {**r!** |→ } • [illegible] .[illegible] . s[illegible] = ∅ .[illegible]r = [illegible] .int[illegible]e-attr[illegible] co[illegible] = ∅

Update_Boundary

Δ**ASDL_System**
$\tau?$: Templates
i-attr : Ports $\rightarrow\!\!\!\!+\!\!\!\rightarrow$ Indices $\rightarrow$ Attributes
$\tau? \notin$ *dom* basis \ Collection
basis($\tau?$).interface-attr = i-attr

Create_PseudoNode

Ξ**ASDL_System**
Δ**ASDL_Unit**
$\tau?$: Templates
$\tau? \notin$ *dom* basis \ Collection
$\exists\, n \in$ Nodes \ *dom* node-parent •
(node-parent' = node-parent $\oplus$ {$n \mapsto \tau?$}
slot-attr' = slot-attr $\oplus$
{(n, p) $\mapsto$ basis($\tau?$).interface-attr(p) | $p \in$ basis($\tau?$).virtual-ports})

3. Conclusions

The increasing scale and complexity of modern software systems requires the development of abstractions than can serve as general organizing principles for architectures. The language described in this paper represents an attempt to do so by giving a coherent and system-independent framework for the formal specification of architectural principles. *Z* has provided a powerful and flexible substrate for the software types and operations of *ASDL*. Other formal approaches to the specification of architectures and architectural styles have been proposed ([1], [2], [5], [6], and [13]). The spirit of this work has much in common with our own ideas. However, we believe that the following features differentiate our work from previous efforts.

First, the software types include a comprehensive set of variable declarations and constraints. These provide expressive power and flexibility, and the *Z* schemas permit the use of a mixture of formal notations for the specification of architectural features. The language therefore supports an approach to architectural specifications that is both heterogeneous and orthogonal. Second, the operations provide a natural way to describe architectural styles that have dynamic features. This approach was illustrated in example (b) where the special event names used in the composition expression referred to the operations *create_node* and *assign_label*. Some approaches (such as [5] and [6]) use languages (like the Chemical Abstract Machine [3] and the π-calculus [8]) that support the dynamic creation of processes. These languages can also be used to model evolving architectures, but they have the drawback of an added semantic complexity when compared to *CSP* and *Z*. In addition, the dynamic operations must be explicitly specified in these languages. In *ASDL*, only a reference to an operation like *create_node* is needed. While *ASDL* currently uses a *Z* schema to specify this operation, another more appropriate formalism could be used.

Finally, the software types and operations provide not only the ability to describe architectural styles, but also a basis for a methodology for developing architectural descriptions. This is analogous to the manner in which abstract data types are used. A stack with its standard operations suggests potential uses in algorithms; a software type with its associated operations suggests potential uses in architectural descriptions. For example, given a module type representing a logical view, to formulate interface requirements for a unit based on the module, it is natural to think in terms of interface operations that specify virtual ports, attributes, links, and

virtual port descriptions. As another example, in describing an architectural style, it is natural to think in terms of whether two nodes belong to the same view and should be connected by an *assign_label* operation, or whether the nodes belong to different views and should be related by an operation that creates entries in a relation between the views. The types and operations of *ASDL* provide a rich variety of choices for handling such issues.

References

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

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

[3] G. Berry and G. Boudol, “The chemical abstract machine model”, Theoretical Computer Science 96 (1992), 217-248.

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

[5] P. Inverardi and A. L. Wolf, “Formal specification and analysis of software architectures using the Chemical Abstract Machine model”, *IEEE Transactions on Software Engineering* **21** (1995), 373-386.

[6] J. Kramer and J. Magee, “Modeling distributed software architectures”, manuscript, 1995.

[7] D. J. Kaplan and R.S. Stevens, “Processing graph method 2.0 semantics”, manuscript, Naval Research Laboratory, June, 1995.

[8] R. Milner, J. Parrow, and D. Walker, “A calculus of mobile processes”, Technical Reports ECS-LFCS 89-85, 89-86, University of Edinburgh, 1989.

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

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

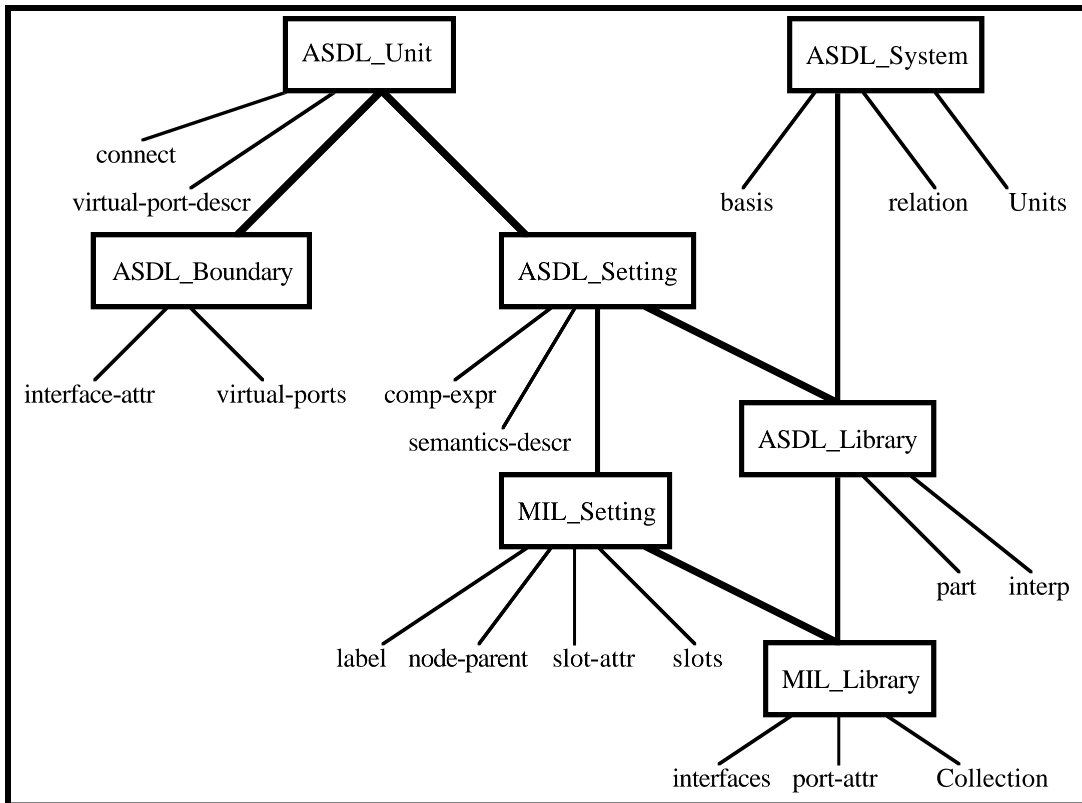
[11] M.D. Rice and S.B. Seidman, “Describing the PGM architectural style”, Technical Report CS-96-120, Department of Computer Science, Colorado State University, 1996.

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

[13] P. Zave and M. Jackson, “Conjunction as composition”, *ACM Trans. on Software Engineering and Methodology*, 2(4), pp. 379-411, Oct. 1993.

Appendix A: Dependencies among ASDL schemas and schema variables

The following graph shows the variables used in the ASDL state schemas and the dependencies between the various schemas. Bold lines indicate schema inclusions and thin lines denote the declaration of variables in schema signatures.



Appendix B [illegible] Sy[illegible] of P[illegible]si[illegible] In[illegible]retations

The syntax is des[illegible]d by u[illegible]re[illegible] no[illegible]n and *BNF*. To focus the presentation, we h[illegible] omitted [illegible]io[illegible]me[illegible]nd b[illegible]ns and standard sequential [illegible]h [illegible]

The members of *Pr[illegible]sExpression[illegible]* [illegible]ib[illegible] the following syntax[illegible]

[illegible] *event* → [illegible] | [illegible] | [illegible]**AIT**(δ) | **STOP** | **SK**[illegible] | *n*

Op ::= **;** | || | ▯ | ⊓

where $n \in$ *Nodes* and δ is a positive integer. The process *event* → [illegible]its the specification of a dynamic composition expression. Events have the following form:

event ::= *operation-name*(*symbol-list*)
operation-name ::= *Create_Node* | *Assign_Label* | ...
symbol ::= *n* | *t* | *l* | *s* | *a*

where $n \in$ *Nodes*, $t \in$ *Templates*, $l \in$ *Labels*, $s = \langle n, p \rangle \in$ *Nodes* × *Ports*, and $a \in$ *SemanticDescriptions*. Each *operation-name* refers to an operation on the system schema (see Appendix A).

Process expressions are therefore constructed from nodes and events using the CSP operators sequential (**;**), parallel (||), deterministic and nondeterministic choice (▯, ⊓), and prefix (→).

Semantic r[illegible]ictio[illegible]lude (i) a[illegible]mpo[illegible] e[illegible]ssion for a setting must use nodes $n \in$ *dom node-paren*[illegible]and sl[illegible] = (*n*, *p*) s[illegible]sfyi[illegible] *i*[illegible]*ces*(*node-parent*(*n*)) and (ii) each *operation-name* has a fixe[illegible]t and order o[illegible]qu[illegible] parameters (e.g. *create_node*(*n*, *t*) and *assign_label*(*l*, *s*, *a*[illegible]Moreover, the co[illegible]n[illegible] operation can modify only the unit variables *connect*, *label*, *nod*[illegible]*ent*, *slot-attr*, an[illegible]*s*.

The members of *In*[illegible]*etatio*[illegible] are desc[illegible] by [illegible]oll[illegible]g syntax:

::= [illegible]*ss* | Op [illegible] (δ[illegible] **STOP** | **SKIP**

where Op is specified above and δ is a positive inte[illegible] Each input-output process has [illegible] following form:

io-process ::= *chan-name* **?** *i-item* → (*i-ite*[illegible] | [illegible]*an-name* **!** *o-item* →
chan-name ::= p | $p^{\mathbf{op}}$ | p_i | $p_i^{\mathbf{op}}$

where $p \in$ *Ports*, *i* is an integer index variable, and **op** is a reserved symbol. The term *i-item* (*o-item*) denotes a variable (expression) for *receiving* (*sending*) data through the port referenced by *chan-name*.

Semantic restrictions include (i) each port name *p* referenced in *interp*(τ) must be a member of *interfaces*(τ), (ii) p or p_i (p^{op} or p_i^{op}) is a legal reference if and only if *port-attr*(*p*).dir = *in* (*port-attr*(*p*).dir = *out*), and (iii) each of the terms *i-item* and *o-item* must must be compatible with the type *port-attr*(*p*).type.