A FORMAL MODEL FOR MODULE INTERCONNECTION LANGUAGES

Technical Report CSE-92-04

M.D. Rice
Mathematics Department
Wesleyan University
Middletown, CT 06459

S.B. Seidman
Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849-5347

July 29, 1992

A Formal Model for Module Interconnection Languages

M. D. Rice

Department of Mathematics
Wesleyan University
Middletown CT 06457 USA

S. B. Seidman

Department of Computer Science and Engineering
Auburn University
Auburn AL 36849 USA

Abstract

A formal, generic model of module interconnections is proposed that is based on the concepts of hierarchy and modularity. The mathematical formalism used in the model is derived from the specification language **Z**. The general form of the model is represented by a collection of **Z** schemas that is invariant across all of its applications. A particular application is specified by supplying values of generic parameters and by conjoining each of the general **Z** schemas with an application-specific **Z** schema and adding additional application-specific constraints. Applications are given to the Conic configuration language, the STILE graphical design and development environment, and to a setting for coarse-grained dataflow programming.

Keywords: module interconnection languages, hierarchy, modularity, specification languages, Z

1. Introduction

It is generally understood that large, complex software systems can be correctly and reliably developed only if their construction has been organized in a hierarchical and modular fashion. As a consequence, the software construction process can be regarded as the assembly of a complex system from a collection of elementary (and hopefully reusable) software components. These components are combined to form composite assemblages, which, in turn, can be regarded as modular units that can be used in further steps in the construction process. Since much of the software construction process reduces to specifying the definition and interconnection of modules, increasing attention is being paid to languages whose primary purpose is to describe module interconnections. Such languages include configuration languages [K90], coordination languages [LS90] and module interconnection languages ([LF85], [G86]). Due to the recent development of these languages, relatively little attention has been paid to the formal description of the module interconnection process. In this paper, a formal, generic model of module interconnections is proposed. The mathematical formalism used in the model is derived from the specification language **Z** ([S88], [S89]). The utility of the model is demonstrated

by showing how it can be used to describe a variety of existing module interconnection languages.

The model is generic in two ways. First, its general form is represented by a collection of **Z** schemas that is invariant across all applications, but which also depend on a number of generic parameters whose values are specific to each particular application. Second, an application is specified by forming the conjunction of each of the general **Z** schemas with an application-specific **Z** schema and adding additional application-specific constraints. Several applications of the model are given in Section 4.

In general, the role of a configuration, coordination, or module interconnection language (we will refer loosely to all of these as MILs) is to express potential data exchanges and synchronizations between computational units (modules). For the purposes of this paper, we assume that an MIL is used in the context of the software development process. In this context, the most important issues are the following, which are closely related to the principles of configuration programming described in [K90]:

(a) <u>A module can be selected from a collection of basic modules</u>
Selection may be viewed either as making a copy of the basic module or as creating a new reference to the basic module. The process can be regarded as a form of renaming, and the basic module can be seen as a template for the new module. Within the formal model presented here, the selection process is called *instantiation* and we say that "a node has been created based on the template". The collection of basic modules will be referred to as the *(template) library*.

(b) <u>Individual modules can be connected to form a module structure</u>
A connection between a pair of modules can be expressed in a variety of ways, including the use of a common parameter in two or more procedure calls and the use of a common edge between two adjacent nodes in a dataflow graph. In the formal model, the potential interfaces of a template will be referred to as *ports*. When these ports are used in the context of a specific renaming (node instantiation), they will be called *slots*. To specify connections between nodes, *labels* will be associated with slots. This idea can be viewed as a generalization of parameter passing. In the special case where each module corresponds to a procedure, a label corresponds to an actual parameter and a slot corresponds to a formal parameter. In this case, the port is represented by the position of the parameter in the parameter list. A *module structure*

is a collection of modules labeled in this way. In the formal model, such a module structure will also be called a *construction setting*.

To illustrate these ideas, consider the following parallel composition of three occam ([I88]) processes, each of which is based on the occam procedure PROC F(CHAN OF INT in, out):

```
PAR
  F(C0, C1)
  F(C1, C2)
  F(C2, C0)
```

In our model, we could view this as follows. The template F is used as the basis for instantiating the nodes $F0$, $F1$, and $F2$. Each node therefore has two slots corresponding to the two ports of F. The label Cj is then assigned to each of the slots $<Fj, in>$ and $<Fk, out>$, where $k = (j-1) \bmod 3$, $j = 0, 1, 2$.

(c) <u>The legality of connections between modules can be checked</u>

This expresses the well-known idea of consistency, which frequently takes the form of type checking in textual languages. More generally, in our model, ports, slots, and labels all have attributes which can be used to enforce consistency requirements. For instance, in the above example, the two ports in F, the corresponding slots in Fj, and the labels Cj would all have the type attribute INT, since the relevant constraint is attribute equality. Attributes for which the labeling constraints require attribute equality will be called *nonstructural*. By contrast, the labeling constraints for *structural* attributes add other requirements (usually involving features of the underlying graph or hypergraph structure) while relaxing equality. In the above example, one possible structural attribute could be based on the direction of the intended data movement. While this idea has no direct analogue in occam, it has been proposed in [R85] as an additional requirement for channel protocols.

(d) <u>A new basic module can be created from a module structure</u>

The creation of a new template from a module structure supports the notion of hierarchy. The encapsulation procedure usually involves some information hiding, so that the new template reflects only the external view of the module structure. In our model, selected slots in a construction setting will be used as the basis for the ports of the new template. Of course, in an actual development process, the internal structure of the application must also be stored. This idea is incorporated into our definition of a *system*.

The model presented here attempts to formalize a large class of methods that have been used (i) to construct module structures while observing constraints imposed by consistency requirements and (ii) to create new basic modules corresponding to a module structure. The construction process uses a renaming procedure that creates modules based on templates. This renaming requires a formalism which summarizes the relationship between the copy and the original while carefully distinguishing the two entities. In particular, the copies are called *nodes*, and the relationship between nodes and their underlying templates is formalized as an *instantiation mapping*. Based on this mapping, *slots* arise naturally as the local (renamed) versions of *ports*. The creation of a new library template based on a module structure requires a formalism for describing the external or visible portion of a module structure. For this purpose, the model provides a formalization which represents an external view of a module structure as a mapping from a subset of selected slots to a set of ports. The ports in the image of this mapping correspond to the ports of the new template, and their attributes are determined by the attributes of the corresponding slots in the module structure.

The model-theoretic specification language **Z** provides a powerful and convenient framework to express the formalism required for the model. The current library and module structure are incorporated into the definition of the state of the construction setting that represents the development process. The construction and encapsulation operations update this state. This approach also permits the formal verification of properties of module interconnection languages.

2. Modular Components in a Construction Setting

Formally, we first assume the existence of two (countably infinite) sets *Templates* and *Ports*. These sets can be regarded as renamings of the natural numbers $\mathbb{N}$. There is also a special nonempty finite set of templates called *Primitives*. The members of *Primitives* are the initial building blocks which are available as components for constructing systems. The assignment of ports to templates will be described by the function *interfaces*. Port attributes are classified by the index sets *Struct* (structural attribute indices) and *NonStruct* (nonstructural attribute indices), and their values are chosen from *Attributes* (the set of attributes). These three sets represent generic parameters which can be specified as needed for a particular application. For convenience, we will define *AttributeIndices* to be the union of *Struct* and *NonStruct*. The assignment of attribute collections to template ports is provided by the function *portattr*.

In general, two node interfaces can be linked only if the attribute values belonging to their associated ports satisfy certain predicates. For example, since nonstructural attributes are used to enforce consistent typing on linked node interfaces, the predicate corresponding to a nonstructural attribute is equality of values. By constrast, the predicates corresponding to structural attributes are application-dependent. This idea will be discussed more completely in Section 4.

We are now ready to give the formal specification of the module library. The *Library* is specified as a **Z** schema, which depends on the generic parameters *Struct*, *NonStruct*, and *Attributes*. The upper half of the schema presents the declarations of the *Library*'s components, while the lower half specifies the constraints that those components must satisfy. In general, we use the **Z** notation described in [S88] and [S89]. A summary of the **Z** notation is given in Appendix B.

Library[Struct,NonStruct,Attributes]

$$
\begin{array}{|l}
\hline
\text{interfaces} : \text{Templates} \rightarrowtail\!\!\!\!\!\!\mid\!\!\rightarrow \mathbb{F}_1\text{Ports} \\
\text{owner} : \text{Ports} \rightarrow\!\!\!\!\!\!\mid\!\!\rightarrow \text{Templates} \\
\text{portattr} : \text{Ports} \rightarrow\!\!\!\!\!\!\mid\!\!\rightarrow \text{AttributeIndices} \rightarrow \text{Attributes} \\
\text{Collection} : \mathbb{F}_1\text{Templates} \\
\hline
\text{Primitives} \subseteq \text{Collection} \\
\text{Collection} = \mathit{dom}\ \text{interfaces} \\
\mathit{disjoint}\ \mathit{ran}\ \text{interfaces} \\
\mathit{dom}\ \text{portattr} = \mathit{dom}\ \text{owner} = \bigcup \mathit{ran}\ \text{interfaces} \\
\forall t : \text{Templates} \bullet t \in \mathit{dom}\ \text{interfaces} \Rightarrow \text{interfaces}(t) = \text{owner}^{-1}(t) \\
\hline
\end{array}
$$

The constraints of the **Library** schema can be paraphrased as follows:

(i) *Collection* is the (nonempty) set of current library templates.

(ii) No two distinct library templates have any ports in common.

(iii) The *owner* function maps a port to its corresponding template.

(iv) The *owner* and *portattr* functions are defined precisely on all ports of all current library templates and only there.

(v) *interfaces* is the inverse of *owner* : *interfaces*(t) is the family of all ports associated with the template t.

The generic parameters *Struct*, *NonStruct*, and *Attributes* that were used in the **Library** schema will appear in all **Z** schemas that describe the formal model. However, we will omit these parameters when referring to the schemas in the text.

Each library template can be incorporated into a construction setting as a node which is selected from the basic set *Nodes*. Just as with *Templates* and *Ports*, *Nodes* can be regarded as a renaming of $\mathbb{N}$. Formally, we write the following **Z** schema, which uses the function *node_parent* to describe the template that corresponds to each node used in the construction setting.

Node_Template_Association[Struct,NonStruct,Attributes]

Library[Struct,NonStruct,Attributes] node_parent : Nodes $\rightarrow\!\!\!\!\!\!\mid$ Templates
ran node_parent $\subseteq$ Collection

The declaration of *node_parent* in this schema implies that a construction setting contains only finitely many nodes. The inclusion of **Library** in the schema declarations implies that all of **Library**'s declarations and constraints, as well as its generic parameters, are also present in **Node_Template_Association.**

We will say that node *n* is *instantiated* based on the template *node_parent*(*n*). When a node *n* is instantiated in a construction setting, it inherits the external interface of *node_parent*(*n*). This is achieved by using the ports associated with *node_parent*(*n*) to define a set of *Slots* that are associated with the node *n*. Thus, in a construction setting, slots are the analogue of ports. These ideas are formalized by the following **Z** schema:

Node_Slot_Association[Struct,NonStruct,Attributes]

Node_Template_Association[Struct,NonStruct,Attributes] Slots : $\mathbb{F}$(Nodes $\times$ Ports)
$\forall(n,p)$: Nodes $\times$ Ports $\bullet$ $(n,p) \in$ Slots $\Leftrightarrow n \in$ *dom*(node_parent) $\wedge$ $p \in$ *dom*(owner) $\wedge$ node_parent(n) = owner(p)

It is easy to show that the preceding scheme is consistent; the predicate in **Node_Slot_Association** insures that there are only a finite number of slots.

For discussing slots, it will be useful to define the projection functions *portproj* and *nodeproj* :

portproj : Nodes $\times$ Ports $\rightarrow$ Ports nodeproj : Nodes $\times$ Ports $\rightarrow$ Nodes
portproj(n,p) = p nodeproj(n,p) = n

The following propositions use these projection functions to show the relationships between the components of the **Node_Slot_Association** schema. The proofs of all propositions will be found in Appendix A.

Proposition 1: **Node_Slot_Association**

$$\vdash node\_parent \circ (Slots \lhd nodeproj) = owner \circ (Slots \lhd portproj).$$

Figure 1 shows the commutative diagram which corresponds to the conclusion of Proposition 1.

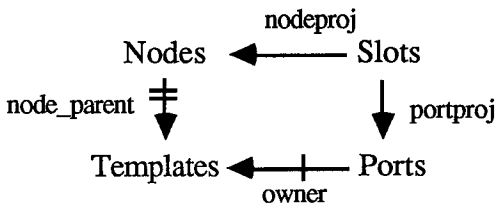


Figure 1: Relations between Slots, Ports, Nodes, and Templates

The following proposition shows that for any set V, any partial function $f : Ports \rightarrow\!\!\!\!\!\!\mid\; V$ that satisfies $dom\ owner \subseteq dom\ f$ can be lifted to a mapping $f^* : Slots \rightarrow\!\!\!\!\!\!\mid\; V$ describing a property of slots, by defining $f^* = f \circ (Slots \lhd portproj)$.

Proposition 2: **Node_Slot_Association** $\vdash ran\ (Slots \lhd portproj) \subseteq dom\ owner$
$\wedge\ ran\ (Slots \lhd portproj) \subseteq dom\ node\_parent.$

One important application of Proposition 2 is the observation that the port attribute function *portattr* can be lifted to a (total) slot attribute function

$$slotattr : Slots \rightarrow AttributeIndices \rightarrow Attributes.$$

The declaration of *slotattr* can therefore be added to the **Node_Slot_Association** schema, along with the predicate $slotattr = portattr \circ (Slots \lhd portproj)$. The revised schema appears below.

The **Z** schemas that have been introduced so far provide a model for an unstructured collection of nodes. In order to specify a structure for this collection, labels will be associated with slots. This will allow a relation on the set of nodes to be defined, where two nodes are related if they have slots with the same label. Formally, we introduce the basic set *Labels* (which can be regarded as a renaming of $\mathbb{N}$), and a partial function $label : Slots \rightarrow\!\!\!\!\!\!\mid\; Labels$. Labels will have associated (nonstructural) attributes, which will be used to check for consistency in labeling the slots that comprise the node interfaces. This is the reason for the introduction of the set *NonStruct*. The attributes of a label are given by a partial function $type: Labels \rightarrow\!\!\!\!\!\!\mid\!\!\mid\; NonStruct \rightarrow Attributes$. A slot can be assigned a label only if the label has been assigned attributes by *type* and if all nonstructural attributes of the slot agree with those of the label. In summary, *NonStruct* represents the

Node_Slot_Association[Struct,NonStruct,Attributes]

Node_Template_Association[Struct,NonStruct,Attributes] Slots : $\mathbb{F}$(Nodes × Ports) slotatttr : Nodes × Ports ⇸ AttributeIndices → Attributes
$\forall(n,p)$: Nodes × Ports • $(n,p) \in$ Slots $\Leftrightarrow n \in dom$(node_parent) $\wedge$ $p \in dom$(owner) $\wedge$ node_parent(n) = owner(p) *dom* slotattr = Slots slotattr = portattr ∘ (Slots ◁ portproj).

set of indices which are used to specify type checking requirements. On the other hand, we will see later that *Struct* represents the set of indices that are used to specify additional requirements on slots needed to define the external interfaces of a construction setting.

The following **Z** schema formalizes the preceding definitions and requirements. This schema corresponds to the description of a construction setting found in the introduction, in which a software system can be specified by defining a collection of related nodes based on primitive templates.

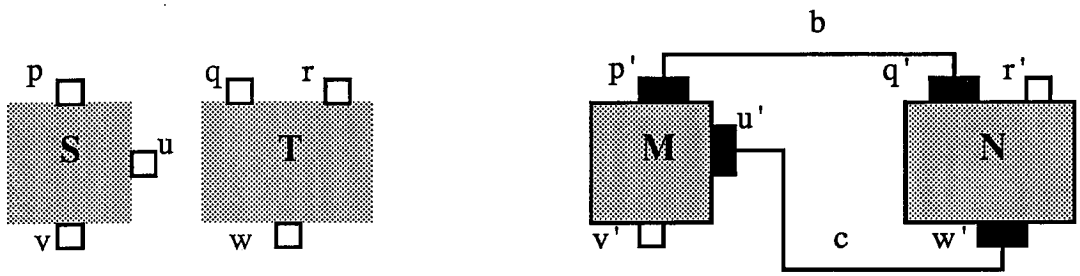
Construction_Setting[Struct,NonStruct,Attributes]

Node_Slot_Association[Struct,NonStruct,Attributes] type : Labels ⇸ [NonStruct → Attributes] label : Nodes × Ports ⇸ Labels
dom label ⊆ Slots *ran* label ⊆ *dom* type $\forall x$: Slots, $\forall j$: NonStruct • $x \in dom$ label ⇒ type(label(x))(j) = slotattr(x)(j)

The preceding schema summarizes those preliminary aspects of the formal model which are needed to specify a single construction setting, where nodes may be instantiated using primitive templates and relationships between nodes may be specified by an appropriate use of labels.

Figure 2 uses a pictorial format to illustrate a library and a construction setting. For simplicity, the attributes of the ports and slots have been suppressed. Note that the lines connecting slots with the same labels are used only for convenience. There is no particular meaning placed on these connections in a general construction setting, except the basic requirement that the corresponding slots have matching non-structural attributes. However, in the context of a specific application, the lines may indicate a potential flow of information between linked nodes. This idea is illustrated in Section 4.

The next section describes some of the fundamental operations provided by the model to modify a construction setting. It will also show how a construction setting can serve as the



Primitives = { S , T }
interfaces(S) = { p,u,v }
interfaces(T) = { q,r,w }

Library

node_parent(M) = S ; node_parent(N) = T
Slots = { p',q',r',u',v',w' }
b = label(p') = label(q') ; c = label(u') = label(w')
{ r',v' } ∩ dom label = ∅

Construction Setting

Figure 2. Library and Construction Setting

basis for a library template and hence for a node in another construction setting, thus adding the concept of hierarchy to the model.

3. Operations in a Construction Setting

Once the *Library* has been initialized, templates can be instantiated as nodes and labels can be assigned to node slots. Each of these actions is accomplished by means of a formally specified operation. Node instantiation uses the *Build* operation, and label assignment uses the *Assign Label* operation. **Z** specifications for these operations are given below.

The *Build* operation is defined formally as

$$\mathbf{Build} = \mathbf{Pre\_Build} \wedge \Xi\mathbf{Library},$$

where the specification for *Pre_Build* is shown below. In this schema, the constraint $n? \notin dom\ node\_parent$ requires, as a precondition of *Build*, that the node $n?$ is currently uninstantiated, while the constraint $node\_parent' = node\_parent \oplus \{n? \mapsto t?\}$ requires that one effect of the *Build* operation is the addition of the ordered pair $(n?, t?)$, corresponding to the instantiation of $t?$ as $n?$, to the function *node_parent*. The other constraints state that the *Build* operation alters neither the *type* function nor the *label* function. Therefore, the three predicates in the schema **Construction_Setting'** are satisfied, so *Pre_Build*, and

hence *Build*, is a legitimate operation. Moreover, the following result shows additional postconditions of the *Build* operation.

Pre_Build[Struct,NonStruct,Attributes]

Δ**Construction_Setting[Struct,NonStruct,Attributes]** $t?$: Templates $n?$: Nodes
$t? \in$ Collection $n? \notin$ *dom* node_parent node_parent' = node_parent $\oplus$ $\{n? \mapsto t?\}$ type' = type $\wedge$ label' = label

Proposition 3: **Build** $\vdash$ $Slots' = Slots \cup \{(n?, p) : Nodes \times Ports \mid owner(p) = t?\}$
$\wedge$ $slotattr' = slotattr \oplus \{(n?, p) \mapsto portattr(p)\}$

In an analogous manner, the *Assign_Label* operation is defined formally as

Assign_Label = Pre_Assign_Label $\wedge$ Ξ**Library,**

where **Pre_Assign_Label** is specified as follows:

Pre_Assign_Label[Struct,NonStruct,Attributes]

Δ**Construction_Setting[Struct,NonStruct,Attributes]** $s?$: Nodes $\times$ Ports $b?$: Labels
$s? \in$ Slots $\setminus$ *dom* label $b? \in$ *dom* type $\forall j$: NonStruct $\bullet$ type($b?$)(j) = slotattr($s?$)(j) label' = label $\oplus$ $\{s? \mapsto b?\}$ type' = type $\wedge$ node_parent' = node_parent

Since the schema **Pre_Assign_Label** includes the constraint *node_parent' = node_parent* and the schema Ξ**Library** includes the constraints *owner' = owner* and *portattr' = portattr*, one can establish the following result:

Proposition 4: **Assign_Label** $\vdash$ $Slots' = Slots \wedge slotattr' = slotattr$.

As in the case of **Build**, the above result states additional constraints of the schema, even if they do not appear explicitly in the specification. Notice that these constraints are not valid for **Pre_Assign_Label**, since its specification does not include the predicate *owner' = owner*.

The current construction setting formalism does not permit the creation of subsystems that can be named, stored, and reused. Consequently, software systems modeled in this way

have a flat structure. By contrast, the structure of a software system that allows the creation and reuse of subsystems is hierarchical. To extend our model to cover such systems, we need to introduce the notion of an *external interface*. The ports of a primitive template may be regarded as representing external interfaces, which are used to define the slots representing the external interfaces of a node instantiated from that template. To create a library template that is based on a construction setting corresponding to a subsystem, the ports comprising the template's external interface must be specified by selecting the slots in the construction setting which will be exported. In general, system designers have substantial freedom to determine the particular slots which will be exported. Formally, we describe the choice of the slots that will be exported from a given construction setting using the following schema:

Interface[Struct,NonStruct,Attributes]

Construction_Setting[Struct,NonStruct,Attributes] selector? : $\mathbb{F}_1$ (Nodes × Ports) interattr: (Nodes × Ports) × [AttributeIndices → Attributes] ⇸ [AttributeIndices → Attributes]
dom label = Slots selector? ⊆ Slots *dom* interattr = {(*s*, slotattr(*s*)) | *s* ∈ selector?} *x* ∈ NonStruct ∧ *s* ∈ selector? ⇒ interattr(*s*, slotattr(*s*)).*x* = slotattr.*x*

In words, the **Interface** schema requires that every slot in the construction setting has been labeled and that at least one slot will be exported. The function *interattr* will be used to determine the structural attributes of the new template port from the attributes of the corresponding slot in *selector?*. If the new template port inherits the structural attributes of the corresponding slot, then *interattr* is simply the projection mapping onto the second coordinate. Nonstructural attributes of a new template port are always inherited from the attributes of the corresponding slot.

A new library entry can now be specified based on the construction setting using the *External* operation, whose schema is shown below. In this schema, the template *t* is the new *Library* entry (*t* ∈ *Collection'*), and the mapping *f* gives the correspondence between the new set of ports for *t* and the slots in the construction setting which have been selected

External[Struct,NonStruct,Attributes]

Interface[Struct,NonStruct,Attributes] **ΔLibrary**
∃*t* ∈ Templates \ Collection ∧ ∃*f* : Ports \ *dom* owner ⤖ selector? • (interfaces' = interfaces ⊕ {*t* ↦ *dom f* } ∧ portattr' = portattr ⊕ interattr ∘ (*f* × (slotattr ∘ *f*))

for export.

The final constraint in **External** says that the interface function *interattr* determines the attributes of every new template port. It is not immediately clear that **External** is a consistent schema, i.e. that the predicates guarantee that only the *Library* portion of the **Construction_Setting** schema is altered. In addition, it is not evident that t and f can be chosen such that the definitions of *interface'* and *portattr'* are valid. We will therefore establish the following result:

<u>Proposition 5</u>: **Interface** $\vdash$ ***pre*** **External.**

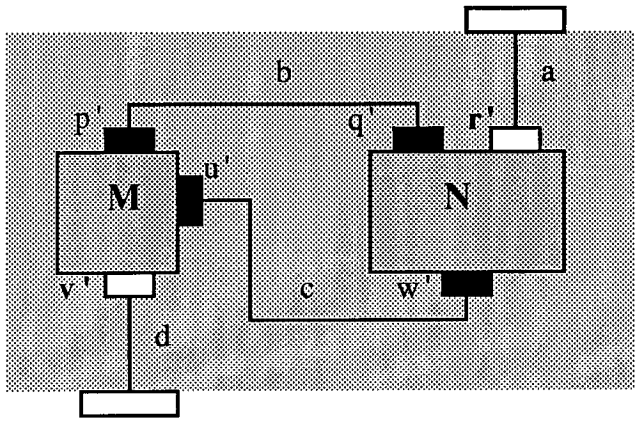
Proposition 5 guarantees that if a selector is chosen which meets the requirements of **Interface** for some predetermined interface function *interattr*, then the construction setting can be used as the basis for a new library template. The slots in the selector essentially become the ports of this template and the attributes of each port are derived from the attributes of the corresponding slot in the construction setting. The restriction that *dom label* = *Slots* is an additional completeness requirement, which says that each port must be labeled before the setting can be packaged as a module.

Once the *Initialize_Library* operation has been performed, a software system is constructed by performing a sequence of *Build*, *Assign_Label*, and *External* operations that instantiate library templates as nodes in a construction setting, link slots on different nodes by labeling, and convert construction settings into new library templates, respectively.

The first two of these operations were used to construct the setting illustrated in Figure 2 based on a **Library** which initially contained the primitive templates S and T. For example, the *Build* operation was used twice to instantiate the nodes M and N based on the templates S and T, respectively. The slot names in the setting correspond to the node-port pairs. For example, p' corresponds to the pair (M, p) and w' corresponds to the pair (N, w). The *Assign_Label* operation was used a number of times with the labels b and c on the slots p', q', u', and w' to obtain the setting shown in Figure 2.

The third operation, *External*, creates a new library template based on a construction setting in which (i) every slot has been labeled and (ii) a selection of slots form the basis for the ports of the new template. This idea is illustrated in Figure 3 using the setting from Figure 2, where the slots r' and v' have been assigned the labels a and d, respectively. The external boxes represent the ports of the new template and the lines connecting these boxes

to the nodes in the construction setting represent the correspondence between the ports and slots.



selector? = { r',v' }

Figure 3. New Library Template

It is clear that although the External operation specifies a new library template, the present formalism does not provide the means for linking the body of the construction setting with the associated template. Furthermore, the formalism is only useful for describing a single construction setting. To create complex hierarchical software systems, the model must be extended by introducing higher-level system schemas to manage the library and the various construction settings. Here we will discuss only the main aspects of system management as described in the **System** schema shown below.

The components of this schema are the *Library* and a collection of mappings which represent various aspects of a software system. The mapping *setting* assigns names to the members of the finite collection of construction settings. The mapping *internal* associates the name of a construction setting with each non-primitive template t; $internal(t) = n$ says that the construction setting $setting(n)$ is the basis for the template t. The mapping *assoc* relates the ports in the non-primitive templates to the slots in the various construction settings: for each non-primitive template t, $assoc(t)$ specifies the correspondence between the ports of t and certain slots in the underlying construction setting $setting(internal(t))$. Finally, the mapping *users* associates each template with the (possibly empty) set of names of construction settings in which a node based on that template was instantiated.

System[Struct,NonStruct,Attributes]

Library[Struct,NonStruct,Attributes] setting: $\mathbb{N}$ –‖–> **Construction_Setting** internal : Templates –‖–> $\mathbb{N}$ assoc : Templates –‖–> [Ports >–‖–> Nodes × Ports] users : Templates –> $\mathbb{F}\mathbb{N}$
dom internal = *dom* assoc = Collection \ Primitives *ran* internal ⊆ *dom* setting $t \in$ Collection \ Primitives ⇒ (*ran* assoc(t) ⊆ setting(internal(t)).Slots ∧ interfaces(t) = *dom* assoc(t) ∧ $p \in$ interfaces(t) ⇒ portattr(p) = setting(internal(t)).slotattr (assoc(t)(p)) ∀ t : Templates • users(t) = {$n \in$ *dom* setting | $t \in$ *ran* setting(n).node_parent }

4. Applications

To describe a particular module interconnection language using our formalism, we will pursue the following general strategy. First, values will be provided for the generic parameters *Struct*, *NonStruct*, and *Attributes*. Second, new schemas will be defined that are based on the generic schemas **Construction_Setting**, **Interface**, **External**, **Build**, and **Assign_Label**, but which incorporate the special characteristics of the application language. In particular, the new schemas describing the state will have the following form:

Specific_State_Schema

Generic_State_Schema **Specific_Signature**
Specific_State_Constraints

where **Specific_Signature** denotes the new signature components and Specific_State_Constraints denotes a corresponding set of constraints on these components that are needed for the language under investigation. In the case of the **Library** schema, these constraints will usually be supplemented by an additional set of Attribute_Constraints involving the attribute functions (*portattr* and *slotattr*).

In a similar manner, the schemas which define new operations on the modified state will have the form shown below. In this case, finitely many **Specific_State_Schema** entries have been added to the **Generic_Operation_Schema**, subject to a possible additional set of Specific_Operation_Constraints. These latter constraints place additional restrictions on the signature components of the **Generic_Operation_Schema**. In general, this set of restrictions will reflect the primary semantic aspects of the particular language.

Specific_Operation_Schema
Generic_Operation_Schema **Specific_State_Schema** **...** **Specific_State_Schema**
Specific_Operation_Constraints

The generality and utility of the approach presented above will be demonstrated by using it to model three rather different MILs: a graphical programming system that was developed in the context of coarse-grained dataflow computation [GRS87], the graphical design and development environment STILE [SW88], and the CONIC environment for the configuration and programming of distributed systems [MKS89].

4.1 A Graphical Programming Environment

The graphical programming environment developed in [GRS87] supports the incremental construction of a certain class of coarse-grain dataflow program graphs. These programs are intended for execution on a special parallel computer which is currently under development [B84]. We will show how our formal model can be specialized to reflect some of the additional constraints needed to construct these program graphs.

The nodes in each graph represent signal processing operations, such as Fourier transforms, convolution, and digital filtering. The edges represent queues which are used to transmit data between nodes. There is also a class of variables which can provide additional information for certain node operations. A typical program graph is shown in Figure 4. Node A represents a primitive signal processing operation, while Node B represents a program graph which has been previously constructed. Qin and $Qout$ are queues which represent an external input and output, respectively, and Qv is an internal queue which is used to transmit data between Nodes A and B. M, N, and P represent input variables.

To apply the modeling strategy to to the construction of program graphs, we must provide appropriate values for the generic parameters *Struct*, *NonStruct*, and *Attributes*, and replace the generic state and operation schemas **Construction_Setting**, **Interface**, **External**, **Build**, and **Assign_Label** by corresponding specific schemas. To specify the generic parameters, each port attribute must be classified as structural or nonstructural, based on its use in linking slots. If two slots can be linked only if their values for a given attribute are identical, that attribute must be classified as nonstructural, while if slots with different values for a given attribute can be linked, then that attribute must be classified as structural.

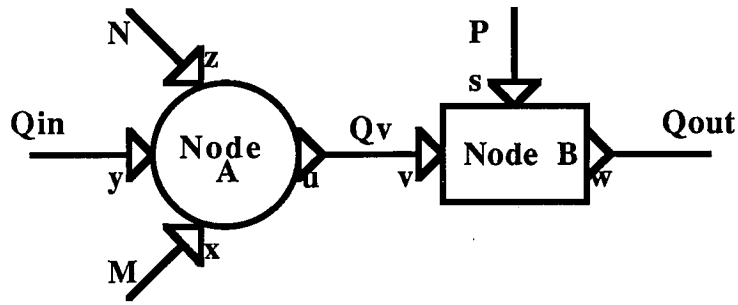


Figure 4 : Program Graph

In this case, the generic parameters are given by *Struct* = {*Dir*} and *NonStruct* = {*Class*, *Data*}. The index *Dir* is used to distinguish an input or output slot, and therefore the associated attribute values are the members of *DirValues* = {*in* , *out*}. The index *Data* is used to specify the datatypes of the queues and variables. For simplicity, in the present description, we will only allow the corresponding attribute values to be the members of *DataValues* = {*float* , *int*}. Finally, the index *Class* is used to describe whether a port or slot refers to a variable or a queue, and the corresponding members of *Attributes* are *ClassValues* = {*queue, var*}.

The preceding remarks represent new constraints in the *Library* schema. In addition, there is an additional constraint that variables can be used only as input parameters. These restrictions are reflected in the following schema:

New_Library[Struct,NonStruct,Attributes]

Library[Struct,NonStruct,Attributes]
$\forall p$: Ports • ($p \in$ *dom* owner $\Rightarrow$ (portattr(p).Data $\in$ DataValues $\wedge$ portattr(p).Dir $\in$ DirValues $\wedge$ portattr(p).Class $\in$ ClassValues $\wedge$ portattr(p).Class = var $\Rightarrow$ portattr(p).Dir = in)

In the construction of program graphs, if two distinct slots of class *queue* have the same label, then a queue connects the respective nodes, so data output from one node will be input to the other. Similarly, if each member of a family of slots of class *var* has the same label, then all the corresponding nodes will receive the same data value. To ensure the correct semantics for sending and receiving data, the labeling is restricted by the following constraint:

$$\text{Label\_Constraint} = x \neq y \in \textit{dom}\ \text{label} \wedge \text{label}(x) = \text{label}(y) \Rightarrow \text{slotattr}(x).\text{Dir} \neq \text{slotattr}(y).\text{Dir} \vee \text{slotattr}(x).\text{Class} = \text{var}$$

The constraint says that distinct slots with the same label either must have different directions or must have class *variable*. Therefore, by *New_Library*, it follows that distinct slots with the same label and different directions must have class *queue*. Using this constraint, we can specify the setting for the construction of the program graphs as follows:

Graph_Construction_Setting[Struct,NonStruct,Attributes]

Construction_Setting[Struct,NonStruct,Attributes] **New_Library[Struct,NonStruct,Attributes]**
Label_Constraint

The methodology for constructing program graphs permits the use of nodes which represent entire subgraphs. These subgraphs are based on graphs which have been constructed in previous graph construction settings and then used as the basis for a new library template. This idea corresponds to the *External* operation discussed in Section 2 except that (i) instead of slots, labels are the basis for the operation and (ii) the labels corresponding to ports in the new template are defined solely in terms of properties of the label function instead of being determined by an auxiliary selection process. This approach can be used because of the additional *Label_Constraint* required in a graph construction setting. Formally, the following sets of labels are defined:

$$\text{External\_Labels} = \{\, b \in \mathit{ran}\ \text{label} \mid (\ \text{type}(b).\text{Class} = \text{queue} \wedge \#\,\text{label}^{-1}(b) = 1\ ) \vee (\ \text{type}(b).\text{Class} = \text{var}\ )\ \}.$$

$$\text{Internal\_Labels} = \{\, b \in \mathit{Labels} \mid \text{type}(b).\text{Class} = \text{queue} \wedge \#\,\text{label}^{-1}(b) = 2\ \}.$$

Based on the preceding definitions, one can show that the range of *label* is precisely the set of External and Internal Labels.

Intuitively, each external label is available for export, that is, it may correspond to a new port in the library template based on the graph construction setting, while each internal label represents a queue which links precisely two distinct slots. The internal labels are not available for export. In order to place these ideas in the framework of the formal model of Section 3, the following constraint defines the selection mechanism solely in terms of the characteristics of the label function. The attributes of the new template ports will be inherited from those of the underlying node slots.

$$\begin{aligned}\text{External\_Constraints} = \\ &\text{selector?} = \text{label}^{-1}(\text{External\_Labels}) \\ &\wedge\ \text{interattr} \circ (\text{id}_{\text{selector?}} \times \text{selector?} \lhd \text{slotattr}) = \text{selector?} \lhd \text{slotattr}\end{aligned}$$

The following schemas can then be used to define the external interface of a graph and the *External* operation:

Graph_Interface[Struct,NonStruct,Attributes]

Interface[Struct,NonStruct,Attributes] **New_Library[Struct,NonStruct,Attributes]**
External_Constraint $\wedge$ Label_Constraint

Graph_External[Struct,NonStruct,Attributes]

External[Struct,NonStruct,Attributes] **ΔNew_Library[Struct,NonStruct,Attributes]**
External_Constraint $\wedge$ Label_Constraint

An analogue of Proposition 5 can also be established, which says that **Graph_Interface** is a precondition of **Graph_External.**

The additional constraint on the label function added in **Graph_Construction_Setting** means that the *Assign_Label* operation described in Section 3 also needs the following additional constraint:

$$\text{Assign\_Constraint} = \forall x{:}\text{Slots} \bullet (\text{slotattr}(s?).\text{Class} = \text{queue} \wedge \text{label}(x) = b?) \Rightarrow \text{slotattr}(x).\text{Dir} \neq \text{slotattr}(s?).\text{Dir}$$

Label assignment in a graph construction setting can be accomplished using the following operation:

Assign_Graph_Label[Struct,NonStruct,Attributes]

Assign_Label[Struct,NonStruct,Attributes] **ΞNew_Library[Struct,NonStruct,Attributes]**
Assign_Constraint $\wedge$ Label_Constraint

4.2 The STILE Graphical Design and Development Environment

For a second application of our modeling approach, we will use the STILE graphical design and development environment [SW88], which allows a system to be described as a hierarchical assemblage of modular components. A STILE system component, called a *part*, is specified by a *catalog page*, which tells what the part does, and a *blueprint*, which tells how the part is constructed. A STILE blueprint is in turn divided into *external* and *internal* sections, which describe the part's external interfaces (*ports*) and implementation, respectively. A blueprint for a composite system is constructed by instantiating blueprints

for the system components. An instantiated blueprint is called a *box*. When a box is instantiated from a blueprint, the implementation contained in the blueprint's internal section is hidden from the user. The ports representing the external interface of the box are determined from the ports of the box components. Blueprints for component parts are added to the blueprint of a composite part that is under construction.

To apply our modeling strategy to STILE, we must provide appropriate values for the generic parameters while replacing the generic state and operation schemas by corresponding specific schemas. For STILE, the attribute classification is $Struct = \{Dir, Spread\}$ and $NonStruct = \{Nature\}$. *Nature* is used to represent an important distinction that STILE makes between data ports and control ports, and therefore the corresponding attribute values are the members of $NatureValues = \{data, control\}$. *Dir* is used to distinguish between input and output ports, so that it corresponds to the members of $DirValues = \{in, out\}$. The attribute *Spread* is used to model the STILE feature that permits catalog pages and blueprints to control the number of links that may be attached to a given port. Since the value of this attribute represents a number of links, $SpreadValues = \mathbb{N}$, the set of natural numbers. Finally,

$$Attributes = DirValues \cup NatureValues \cup SpreadValues.$$

Formally, all of these requirements can be specified by the **STILE_Library** schema :

STILE_Library[Struct,NonStruct,Attributes]

Library[Struct,NonStruct,Attributes]
$\forall p : \text{Ports} \bullet (p \in dom\ \text{owner} \Rightarrow$
$(\ \text{portattr}(p).\text{Dir} \in \text{DirValues}$
$\wedge\ \text{portattr}(p).\text{Nature} \in \text{NatureValues}$
$\wedge\ \text{portattr}(p).\text{Spread} \in \text{SpreadValues})$

In STILE, the association of box ports with opposite orientation attributes and identical data/control attributes is dependent on the availability of sufficient link capacity, as modeled by the *Spread* attribute. To verify this, the current partners of a given node slot must be identified. This is specified by the *partners* function in the **STILE_Links** schema shown below, where the partners of a given slot share the label of the slot, but have opposite direction.

STILE_Links[Struct,NonStruct,Attributes]

Node_Template_Association[Struct,NonStruct,Attributes] slotattr.Dir: Nodes × Ports ⇸ Attributes partners: Nodes × Ports → $\mathbb{F}$(Nodes × Ports)
dom slotattr.Dir = Slots $\forall s$: Nodes × Ports • ($s \in$ Slots ⇒ slotattr.Dir(s) = slotattr(s).Dir) $\forall s$: Nodes × Ports • $s \in$ *dom* label ∧ partners(s) = label^{-1}(label(s)) \ slotattr.Dir^{-1}(slotattr.Dir(s)) ∨ ($s \notin$ *dom* label ∧ partners(s) = ∅)

Once this has been done, the **Construction_Setting** schema can be modified to verify the existence of the necessary link capacity:

STILE_Construction_Setting

Construction_Setting **STILE_Library ∧ STILE_Links**
Spread_Constraint

where Spread_Constraint =

$$\forall s : \text{Nodes} \times \text{Ports} \bullet s \in \text{Slots} \Rightarrow (\#\text{partners}(s) \leq \text{slotattr}(s).\text{Spread})$$

This new constraint requires that the number of slots linked to a given slot by a shared label is bounded by that slot's value of the *Spread* attribute. Note that since *Nature* is a nonstructural attribute, linked box ports must share the same value for this attribute.

The *External* operation specified in the preceding section converts a construction setting into a library template. The blueprint corresponding to the new template can be used in the construction of other, more complex parts. The external portion of the blueprint is obtained from the external portions of the component blueprints, which must therefore be recognized by the STILE version of *External*. In particular, the externally visible slots of a STILE box are those whose linkage capacity, as represented by *Spread*, is not fully used. Formally, this can be specified by

STILE_Interface

Interface **STILE_Library ∧ STILE_Links**
Spread_Constraint Interface_Constraint

where Interface_Constraint =

$$\text{selector?} = \{s \in \text{Slots} \mid \#\text{partners}(s) < \text{slotattr}(s).\text{Spread}\}$$
$$\wedge\ (\forall s : \text{Nodes} \times \text{Ports} \bullet s \in \text{selector?} \Rightarrow$$

$$\text{interattr}(s,\text{slotattr}(s)).\text{Dir} = \text{slotattr}(s).\text{Dir}$$
$$\wedge\ \text{interattr}(s,\text{slotattr}(s)).\text{Spread} = \text{slotattr}(s).\text{Spread} - \#\text{partners}(s)).$$

Notice that the linkage capacity of each resulting template port has been reduced from its original capacity by the amount that it has been used within its underlying construction setting. Thus in this application, by contrast with the dataflow application, the structural attribute values of template ports have been changed from their original values.

STILE_External can now be specified as follows:

STILE_External

External **ΔStile_Library ∧ ΞStile_Links**
Spread_Constraint Interface_Constraint

The constraints imply that the value of *Spread* for the new template port is equal to the unused link capacity of the underlying node slot.

The *Build* operation that has been specified in Section 3 can also be used here to instantiate nodes that correspond to STILE boxes that are assembled from existing blueprints. Formally, we will specify *STILE_Build* as

STILE_Build

Build **ΞSTILE_Library ∧ ΞSTILE_Links**
Spread_Constraint

The specification of a complex STILE part can now be formalized as a sequence of *STILE_Build* operations.

It is now relatively easy to specify *STILE_Assign_Label*:

STILE_Assign_Label

Assign_Label **ΞSTILE_Library ∧ ΔSTILE_Links**
Spread_Constraint STILE_Assign_Constraint

where STILE_Assign_Constraint $= \#\text{partners}(s?) < \text{Slotattr}(s?).\text{Spread}$.

This requirement guarantees that the constraints of **STILE_Construction_Setting** are maintained, so that **STILE_Assign_Label** is a consistent schema.

So far, we have seen that the formalism presented in this paper meets the requirements of STILE quite naturally. There is one important STILE feature, however, that does not fit easily into the framework that we have developed. In STILE, each box must be given external and internal *interpretations,* which determine the semantics that will be used to govern the execution of the box. For the purposes of this paper, we need not discuss the choices in more detail [SW88]; it suffices to say that the external interpretations of all boxes used in the construction of a composite box must match. When the construction is complete (i.e. when *External* is invoked), the user must choose the external interpretation of the new composite box, while its internal interpretation is the same as the (identical) external interpretations of its components. In order to incorporate this feature into our model, the formalism will have to be extended to include template and node attributes which can be regarded as describing potential or actual node semantics. Node semantic attributes will be inherited from the underlying template. When *External* is invoked to construct a new template, some of the template's semantic attributes will be determined by the semantic attributes of its components, while others will be determined by the user in a way similar to that used for the *Spread* attribute. The necessary extensions retain the spirit of the formalism presented above, and they will not be described here.

4.3 The CONIC Configuration Language

Finally, we will use our approach to model the CONIC environment for the specification of distributed systems. The CONIC environment provides a programming language to describe computations carried out by the lowest-level components of a distributed system, and also a configuration language that indicates how those components are assembled in a system. We will use our ideas to provide a formal model of the way CONIC's configuration language permits systems and system components to be assembled from simpler components. The programming of a system's lowest level components will not be considered in this discussion.

Systems specified using the CONIC configuration language are assemblages of *group modules*, which are hierarchical entities composed of smaller modules. These modules may be other group modules, or they may be *task modules*, which comprise the lowest hierarchical level and are specified using the CONIC programming language. The external interfaces of CONIC modules are provided by typed *entryports* and *exitports*, which

```
group module Monitor(Tfactor, Pfactor:integer);
        exitport
                press, temp : real reply signaltype;
        entryport
                control : boolean;
        use
                 scale: sensor;
        create
                Temperature : sensor;
                Pressure : sensor;
                Tscale : scale(Tfactor);
                Pscale : scale (Pfactor);
        link
                Temperature.output to Tscale.output;
                Pressure.output to Pscale.input;
                Tscale.output to temp;
                Pscale.output to press;
                control to Tscale.control, Pscale.control;
        end.
```

Figure 5. CONIC Program for Monitor

consist of two classes: *request-reply* ports and *notify* ports. Communication between CONIC modules is specified by linking ports in their external interfaces that have the same type and class. While a single *notify* exitport may be linked to a set of *notify* entryports (or the reverse), a *request-reply* exitport may only be linked to a single *request-reply* entryport.

A typical CONIC system configuration is illustrated by the example shown in Figure 5 and

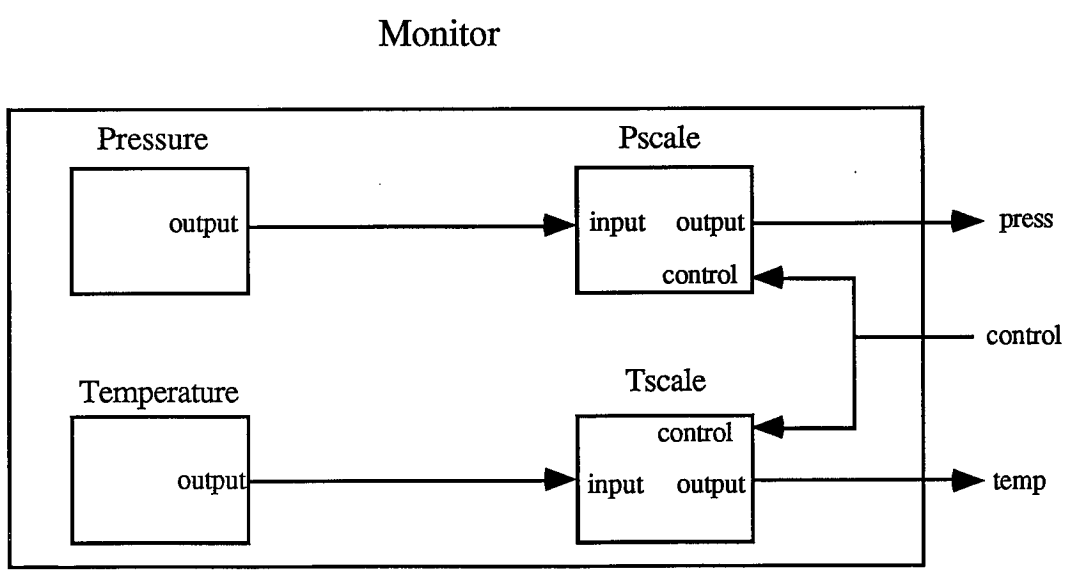


Figure 6. Monitor Group Module

illustrated in Figure 6, which is taken from [MKS89]. Notice that *press* and *temp* are external request-reply interfaces of *Monitor*, while *control* is a external notify interface, which is linked to two task module (notify) interfaces. The definition of the group module *Monitor* uses the task modules *scale* and *sensor*. The CONIC programs that define *scale* and *sensor* also determine the external interfaces of these task modules.

To apply our modeling strategy to CONIC, we must provide appropriate values for the generic parameters while replacing the generic state and operation schemas by corresponding specific schemas. For CONIC, the attribute classification is $Struct = \{Class, Category\}$ and $NonStruct = \{Mode, LinkType\}$. *Class* is used to distinguish between entryports and exitports, and therefore the corresponding attribute values are the members of $ClassValues = \{entryport, exitport\}$. *Category* is used to distinguish between ports that represent external interfaces of templates being designed in a top-down fashion and ports that represent external interfaces of primitive or completely assembled templates. The corresponding attribute values are the members of $CategoryValues = \{top\_down, bottom\_up\}$. *Mode* is used to distinguish between notify and request-reply ports, so that it corresponds to the members of $ModeValues = \{notify, request\text{-}reply\}$. *LinkType* is used to describe the type of an external interface, and the corresponding values are members of a set *LinkTypeValues*, which contains all types provided by the CONIC system. Finally,

$$Attributes = ClassValues \cup ModeValues \cup CategoryValues \cup LinkTypeValues.$$

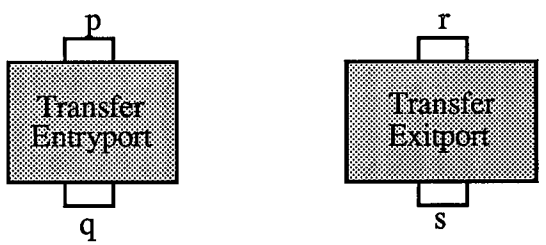
It is important to note that the external interfaces of a group module are *supplied explicitly* by the **exitport** and **entryport** statements in its CONIC specification, as illustrated by the statements

exitport press, temp : real **reply** signaltype
entryport control : boolean

found in the CONIC example presented above. To formalize this idea in our model, we will assume that the *Primitive* templates in the *Library* contain a collection of *Transfer* templates. Each *Transfer* template corresponds to a single group module entryport or exitport. A *Transfer* template has two ports which have the *LinkType* and *Mode* attribute values that correspond to the desired interface, but which have opposite values on the *Class* and *Category* attributes. In particular, if a module interface is intended to be an exitport, then the corresponding *Transfer* template has two ports, whose values on (*Class*,

Category) are (*exitport, top_down*) and (*entryport, bottom_up*). If the interface is intended as an entryport, the occurrences of *top_down* and *bottom_up* are reversed. This idea is illustrated in Figure 7.

Transfer templates were not needed in each of the first two applications of our approach. This was the case because both the dataflow graph environment and the STILE environment use an exclusively bottom-up design strategy. As a consequence, the external interfaces of a construction setting can be completely inferred from the interfaces of its constituents. CONIC uses both top-down and bottom-up design strategies, and the transfer templates are needed to incorporate the explicit external interfaces of modules that are specified in a top-down fashion.



portattr(p).Class = entryport	portattr(r).Class = exitport
portattr(q).Class = exitport	portattr(s).Class = entryport
portattr(p).Category = top_down	portattr(r).Category = top_down
portattr(q).Category = bottom_up	portattr(s).Category = bottom_up

Figure 7. Transfer Templates

Formally, all of these requirements are specified by the **Conic_Library** schema.

Conic_Library[Struct,NonStruct,Attributes]

Library[Struct,NonStruct,Attributes]
transfer : LinkTypeValues × ClassValues × ModeValues → Primitives

$\forall p$: Ports • ($p \in$ *dom* owner ⇒
(portattr(p).Class ∈ ClassValues
∧ portattr(p).Category ∈ CategoryValues
∧ portattr(p).Mode ∈ ModeValues
∧ portattr(p).LinkType ∈ LinkTypeValues)

$\forall (v_1, v_2, v_3)$: LinkTypeValues × ClassValues × ModeValues •
(interfaces(transfer(v_1,v_2, v_3)) = $\{p_1, p_2\}$
∧ portattr(p_k).LinkType = v_1 ∧ portattr(p_k).Mode = v_3, $k = 1, 2$
∧ portattr(p_1).Category = top_down ∧ portattr(p_2).Category = bottom_up
∧ portattr(p_1).Class = v_2 ∧ portattr(p_2).Class ≠ v_2)

In CONIC, a linkage can be created either between an exitport and an entryport of component modules, each of which forms part of a group module's specification (as in the linkage of *Temperature.output* to *Tscale.input*), or between an entryport (exitport) of such a module and an entryport (exitport) of the module that is currently being specified (as in the linkage of *Tscale.output* to *temp*). These possibilities can be formally specified by the following **Z** schema:

Conic_Sink_Source[Struct,NonStruct,Attributes]

Node_Template_Association[Struct,NonStruct,Attributes] SINKS : $\mathbb{F}$(Nodes × Ports) SOURCES: $\mathbb{F}$(Nodes × Ports) Sinks : Labels → $\mathbb{F}$(Nodes × Ports) Sources : Labels → $\mathbb{F}$(Nodes × Ports)
SINKS = {s ∈ Slots | (slotattr(s).Class = *entryport* ∧ slotattr(s).Category = *bottom_up*)} SOURCES = {s ∈ Slots | (slotattr(s).Class = *exitport* ∧ slotattr(s).Category = *bottom_up*) } Sinks(b) = label^{-1}(b) ∩ SINKS Sources(b) = label^{-1}(b) ∩ SOURCES

We can now specify the structure of a CONIC construction setting as follows, using the format described in the preceding section:

Conic_Construction_Setting

Construction_Setting **Conic_Library ∧ Conic_Sink_Source**
Specific_Constraints$_{CS}$

where Specific_Constraints$_{CS}$ =

$$\forall\, s,t : \text{Nodes} \times \text{Ports},\ \forall\, b : \text{Labels} \bullet$$
$$s \in \text{label}^{-1}(b) \Rightarrow (\ \text{slotattr}(s).\text{Mode} = \textit{notify} \wedge (\#\text{Sources}(b) \le 1 \vee \#\text{Sinks}(b) \le 1))$$
$$\vee\ (\ \text{slotattr}(s).\text{Mode} = \textit{request-reply} \wedge (\#\text{Sources}(b) \le 1 \wedge \#\text{Sinks}(b) \le 1))$$
$$\wedge\ ((s \ne t\ ) \wedge (\text{label}(s) = \text{label}(t))) \Rightarrow (\text{slotattr}(s).\text{Category} = \textit{bottom\_up} \wedge \text{slotattr}(t).\text{Category} = \textit{bottom\_up})$$

The first of the new constraints states that at least one of the sets of slots linked by a shared label has at most one member. The second constraint states that a reply-request slot can be linked to at most one other reply-request slot, and that if two reply-request slots are linked,

they must have different Class attribute values. The final constraint states that two slots that have been specified as external slots can not be linked together. Note that since *Mode* and *LinkType* are nonstructural attributes, linked module slots must share the same attribute values for these attributes.

The *External* operation specified in the preceding section converts a construction setting into a library template. In the CONIC context, the template resulting from an *External* operation performed on a group module can be used to include an instance of that group module in the construction of another group module. Since the external interfaces of a group module are explicitly supplied in the CONIC specification, they must be recognized by the CONIC version of *External*. Formally, this can be done by first specifying

Conic_Interface

Interface **Conic_Library** $\wedge$ **Conic_Sink_Source**
$\text{Specific\_Constraints}_{CS}$ $\text{Specific\_Constraints}_{I}$

where

$$\begin{aligned}\text{Specific\_Constraints}_{I} = \text{selector?} = \{s \in \text{Slots} \mid \text{slotattr}(s).\text{Category} = \textit{top\_down} \\ \wedge\, \text{node\_parent}(\text{nodeproj}(s)) \in \textit{ran}\ \text{transfer}\} \\ \wedge\ (\forall s \in \text{selector?} \bullet \\ (\text{interattr}(s, \text{slotattr}(s)).\text{Class} = \text{slotattr}(s).\text{Class} \\ \wedge\, \text{interattr}(s, \text{slotattr}(s)).\text{Category} = \textit{bottom\_up})\ )\end{aligned}$$

The new constraint defines the externally visible slots to be precisely those slots that correspond to the external interfaces specified for the group module, and also states that while the Class attribute value for each port of the new template is inherited from the Class attribute value for the corresponding slot, its Category attribute value is always equal to *bottom_up*.

Conic_External can now be specified as follows:

Conic_External

External Δ**Conic_Library** $\wedge$ Ξ**Conic_Sink_Source**
$\text{Specific\_Constraints}_{CS}$ $\text{Specific\_Constraints}_{I}$

Note that no additional specific constraints are needed, and further that the specification of *selector?* in **Conic_Interface** guarantees that the ports representing the external interfaces of the resulting template have the value *bottom_up* on the *Category* attribute, so that they can be used as external interfaces of a module that is a component in a further construction process.

The *Build* operation that was specified in Section 3 can be used here to instantiate nodes that are based on instances of task modules or group modules that have already been assembled. Formally, we will specify *Conic_Build* as

Conic_Build

Build **ΞConic_Library $\wedge$ ΞConic_Sink_Source**
$\text{Specific\_Constraints}_{CS}$

The creation of the external interface of a group module construction setting can now be formalized as a sequence of *Conic_Build* operations that instantiate templates that are members of the range of the *transfer* function, with one operation needed for each individual interface.

It is now relatively easy to specify **Conic_Assign_Label**:

Conic_Assign_Label

Assign_Label **ΞConic_Library $\wedge$ ΔConic_Sink_Source**
$\text{Specific\_Constraints}_{CS}$ $\text{Specific\_Constraints}_{AL}$

$$
\begin{aligned}
\text{where Specific\_Constraints}_{AL} = \ \neg\ (\ & (\ s? \in \text{SOURCES} \wedge \#\text{Sources}(b?) = 1 \\
& \wedge (\text{slotattr}(s?).\text{Mode} = \textit{notify} \Rightarrow \#\text{Sinks}(b?) \geq 2\ )) \\
& \vee (\ s? \in \text{SINKS} \wedge \#\text{Sinks}(b?) = 1 \\
& \wedge (\text{slotattr}(s?).\text{Mode} = \textit{notify} \Rightarrow \#\text{Sources}(b?) \geq 2))) \\
& \wedge\ \forall t : \text{Slots} \bullet t \in \text{label}^{-1}(b?) \Rightarrow \\
& \quad \text{slotattr}(t).\text{Category} = \textit{bottom\_up} \\
& \wedge\ \text{slotattr}(s?).\text{Category} = \textit{top\_down} \Rightarrow \\
& \quad \text{label}^{-1}(\{b?\}) = \varnothing
\end{aligned}
$$

This constraint guarantees that the constraints of **Conic_Construction_Setting** are maintained, so that **Conic_Assign_Label** is a consistent schema.

5. Conclusions

The present paper has outlined a general model that can be used to describe a process by which hierarchical and modular software systems can be built. The restrictions presented in the various **Z** schemas comprise a construction semantics for the model. The deliberate intention was to impose minimal semantics in order to obtain a high degree of generality. Although most applications will require additional semantics, the minimal semantics of the model are invariant across all applications and the basic semantic restrictions will not be superseded by the new requirements.

Note that we have not described any execution semantics for our model. It would be difficult to incorporate such semantics without also restricting the range of potential applications. Nonetheless, in specific situations, it appears that our construction semantics will support the addition of execution semantics. For example, a CCS model [M89] for the execution semantics may be applicable, since our use of labels corresponds naturally to the use of names and co-names in CCS.

The formalism presented in this paper evolved from the work on the program graph environment described in the preceding section. The applications to Conic and STILE demonstrate that it can serve as a framework for the ideas expressed by a variety of systems. The most natural applications involve systems which have a modular and hierarchical orientation and are based on a bottom-up construction philosophy. However, as we saw in the Conic application, the formalism can also be used to model certain top-down systems, including the modular design techniques proposed in [RY88]. The crucial observation is that top-down design systems are modeled using additional transfer templates to circumvent the requirement that each port on a user-defined template obtains its attributes from some slot in the corresponding construction setting. Additional attributes can then be used to distinguish top-down and bottom-up design features.

Clearly, the operations chosen for illustration in this paper do not include all operations which are desirable in a given application. For example, deletion operations for both labels and nodes will usually be needed. Moreover, in an application setting, the basic operations will have to be modified to permit the updating of the **System** schema defined in Section 3. In addition, other operations will be needed to manage the various construction settings and the addition of user-defined templates to the *Library*.

Appendix A: Proofs of Propositions

Proposition 1 : Choose $(n,p) \in$ Slots. By definition, nodeproj$(n, p) = n$ and portproj$(n, p) = p$. Therefore, based on the definition of Slots in Node_Slot_Association, one can write

$$\begin{aligned} \text{node\_parent(nodeproj}(n,p)) &= \text{node\_parent}(n) \\ &= \text{owner}(p) \\ &= \text{owner(portproj}(n, p)). \end{aligned}$$

[]

Proposition 2 : Choose $x \in ran$ (Slots $\lhd$ portproj). By definition, there exists $(n,p) \in$ Slots such that portproj$(n, p) = x$. By definition of portproj, $x = p$, so by the definition of Slots, $x = p \in dom$ owner. A similar argument establishes that ran (Slots $\lhd$ nodeproj) $\subseteq dom$ node_parent. []

Proposition 3 : Based on Ξ**Library**, owner' = owner. Since node_parent' is an extension of node_parent, it follows by definition that Slots $\subseteq$ Slots'. Conversely, assume $(n, p) \in$ Slots'. If $n \in dom$ node_parent, then by definition, $(n, p) \in$ Slots, so assume that $n \notin dom$ node_parent. Then $n \in dom$ node_parent' $\Rightarrow n = n?$, so by definition, node_parent'$(n?) = t?$ = owner'(p) = owner(p). Hence Slots' $\subseteq$ Slots $\cup$ $\{(n?, p)$: owner$(p) = t?\}$. On the other hand, if owner$(p) = t?$ for some port p, then node_parent'$(n?) = t?$ = owner'(p), so by definition, $(n?, p) \in$ Slots'. This establishes the first equality. The second equality follows from the first equality based on the definitions of slotattr and slotattr'. []

Proposition 4 : Based on **Pre_Assign_Label**, node_parent' = node_parent, and based on Ξ**Library**, owner' = owner. Therefore, by definition, Slots' = Slots. In addition, portattr' = portattr from Ξ**Library**. To establish slotattr' = slotattr, note that slotattr' = portattr' $\circ$ (Slots' $\lhd$ portproj) = portattr $\circ$ (Slots $\lhd$ portproj) = slotattr. []

Proposition 5 : Since **Library** is included in **Construction_Setting** which, in turn, is included in **Interface**, the signature of **External** is satisfied by **Interface**. Therefore, it suffices to show (i) how to choose mappings t and f so that the predicates in **External** hold, (ii) that a schema **Library'** has been specified, and (iii) that only the **Library** portion of **Interface** is altered by **External**.

Based on **Library**, *Collection* = *dom* interfaces is a finite set. Since *Templates* is an infinite set, it follows that one can choose $t \in$ *Templates* $\setminus$ *Collection*. Also, based on **Library**, *dom* owner = $\cup$ *ran* interfaces. Since *dom* interfaces is a finite set and every set

in the range of interfaces is finite, it follows that *dom* owner is also a finite set. Hence $Ports \setminus dom$ owner is an infinite set. Therefore, based on **Interface**, one can select a subset $P \subseteq Port \setminus dom$ owner such that $\#P$ = #selector? and select a bijection $f : P \rightarrowtail$ selector?. This completes the proof of (i).

To establish (ii), we must show that the signature and the predicates in **Library'** are satisfied when interfaces' and portattr are defined as in **External**. To establish the predicates, one must show (a) *disjoint ran* interfaces' and (b) *dom* portattr' = *dom* owner' = $\cup$ *ran* interfaces'. To prove (a), note that *disjoint ran* interfaces and *dom* owner = $\cup$ *ran* interfaces hold by assumption. Therefore, *dom f* $\cap$ *dom* owner = $\varnothing \Rightarrow$ *dom f* $\cap$ interfaces(t) = $\varnothing$ for every $t \in$ *dom* interfaces, so *disjoint ran* interfaces' also holds. To prove (b), by assumption, *dom* portattr = *dom* owner = $\cup$ *ran* interfaces. By definition, *dom* portattr' = *dom* portattr $\cup$ *dom f*. Therefore, by definition,

$$\begin{aligned} \textit{dom}\ \text{owner}' &= \textit{dom}\ \text{owner} \cup \textit{dom}\, f \\ &= \textit{dom}\ \text{portattr} \cup \textit{dom}\, f = \textit{dom}\ \text{portattr}' \\ &= \cup\ \textit{ran}\ \text{interfaces} \cup \textit{dom}\, f = \cup\ \textit{ran}\ \text{interfaces}'. \end{aligned}$$

This establishes part (b). To show that the signature in **Library** is satisfied, by (i) *dom f* is a finite set, so both interfaces' and portattr' satisfy the signature of **Library'**. By part (a), *dom* owner' = *dom* portattr', so owner' also satisfies the signature of **Library'**. Finally, since *Collection'* = *dom* interfaces' and contains *Collection*, *Collection'* is a non-empty finite set ; therefore, it satisfies the signature of **Library**.

Finally, to establish (iii), we must show that node_parent' = node_parent, type' = type, label' = label, Slot' = Slots, and slotattr' = slotattr. The first three equalities hold since these entities can be altered only by direct definition. This is not the case for the last two entities since their definitions involve owner' and portattr'. To establish Slots' = Slots, note that Slots $\subseteq$ Slots' since owner' is an extension of owner and node_parent' = node_parent. Conversely, choose $(n, p) \in$ Slots'. By definition, $p \in$ *dom* owner', $n \in$ *dom* node_parent', and owner'(p) = node_parent'(n) = node_parent(n). If $p \notin$ *dom* owner, then by definition of interfaces', $p \in$ *dom f*, so node_parent(n) = owner'(p) = t. Based on **Node_Template_Association**, *ran* node_parent $\subseteq$ *Collection*, so it follows that $t \in$ *Collection*, which contradicts the choice of $t \notin$ *Collection*. Hence $p \in$ *dom* owner, so owner(p) = owner'(p) = node_parent(n) $\Rightarrow$ $(n, p) \in$ Slots. Therefore, Slots' = Slots. A proof identical to the one given in Proposition 4 establishes slotattr' = slotattr. This completes the proof of part (iiii). []

Appendix B: Z Notation

Z is a specification language which has been developed at Oxford University. References to **Z** include a reference manual [S89], a collection of case studies [H87] and the formal semantics of **Z** [S88]. This appendix gives a brief introduction to **Z** schema notation.

(i) Schema Notation

Z is based on typed set theory and uses *schemas* to define functions and types. A schema associates definitions of typed variables with predicates that constrain their possible values. The simplest variable types name familiar sets such as the natural numbers $\mathbb{N}$. More complex types are built using type constructors which are analogues of familiar set operations: power set formation ($\mathbb{P}$), products ($\times$), and function space formation ($\rightarrow$).

The first kind of schema defines one or more functions, and has the form

```
| declaration(s)
|----------------
| predicate(s)
```

This is called an *axiomatic definition*. It introduces global constants which can be used in later **Z** schema definitions.

The second kind of schema assigns a name to a group of variable declarations and predicates relating these variables. This is called a *schema definition*, and has the form

```
  S
|----------------
| declaration(s)
|----------------
| predicate(s)
|----------------
```

A schema name S can be used as a type, and the declaration $w : S$ declares a variable w whose components are declared in S. For example, if x is a variable declared in S, then $w.x$ denotes the x component of w. A schema definition also may use *generic parameters* $X_1, X_2, ..., X_n$ associated with the schema name: $S[X_1, X_2, ..., X_n]$. These parameters are set constants which can be used in the schema definition.

A schema S can be included in the declarations of another schema T, in which case the declarations of S are merged with the other declarations of T and the predicates of S and T are conjoined. An inclusion has the form

```
  T
|----------------
| S
| declaration(s)
|----------------
| predicate(s)
|----------------
```

(ii) Schema Operations

The schemas formed by merging the declarations of schemas S and T are denoted by

$S \wedge T$	if the respective predicates are conjoined
$S \vee T$	if the respective predicates are disjoined

Schema definitions may be used to specify *operations* on a state specified by another schema. In this case, the following conventions are used for variable names:

undashed	state before
dashed (')	state after
ending in ?	inputs
ending in !	outputs

Given a schema S,

S'	schema obtained by using dashes to rename all components declared in S
pre S	schema obtained by hiding all declarations of S which are dashed(') or outputs (ending in !) (If any such components are used in predicates, they become existentially quantified in *pre* S.)

If the components of a schema S are used and altered by an operation represented by the schema O, then the schema $\Delta S = S \wedge S'$ is included in the schema O. If the operation uses the components of S without altering them, then the schema ΞS is included in O.

(iii) Set Notation

$\mathbb{N}$	set of natural numbers {0, 1, ... }

Given a set S,

#S	cardinality of the set S
$\mathbb{P}$S ($\mathbb{F}$S)	set of all subsets (all finite subsets) of S
$\mathbb{P}_1$S ($\mathbb{F}_1$S)	set of all non-empty subsets (all non-empty finite subsets) of S

Given G $\subseteq$ $\mathbb{P}$S,

$\cup$ G	union of all subsets in the family G
disjoint G	predicate which is true if and only if G is a pairwise-disjoint family

Given sets S and T,

$$S \times T = \{(x,y) : x \in S \wedge y \in T\}$$
$$S \setminus T = \{ x \in S : x \notin T\}$$

(iv) Logic Notation

$\exists x : T \bullet P$	there exists x of type T such that P holds
$\forall x : T \bullet P$	for all x of type T, P holds
$\{ x : T \mid P \}$	set of all x's of type T such that P holds

Assume that S is a schema and P is either a schema or a collection of predicates.

$S \vdash P$	the predicates associated with P can be deduced from S

(v) Function Notation

Given sets S and T,

$[S \rightarrow T]$		set of all *total* functions from S to T
$[S \rightarrow\!\!\!\!\mid T]$		set of all *partial* functions from S to T
$[S \rightarrowtail T]$	$(S \rightarrowtail\!\!\!\!\mid T)$	one-to-one total (partial) functions
$[S \twoheadrightarrow T]$	$(S \rightarrowtail\!\!\!\!\mid\!\!\twoheadrightarrow T)$	onto total (partial) functions
$S \rightarrow\!\!\!\!\mid\!\!\mid T$		partial functions with finite domains
id_S		the identity function on the set S

Note that $\rightarrow$ (and $\rightarrow\!\!\!\!\mid$) are right associative operators. For example, $f : S \rightarrow T \rightarrow V$ is shorthand for $f : S \rightarrow [T \rightarrow V]$, i.e. for each $x \in S$, $f(x) : T \rightarrow V$. In this case, we write $f(x).y$ instead of $f(x)(y)$.

Given $f : S \rightarrow\!\!\!\!\mid T$,

$dom\, f = \{x \in S : f(x) \text{ is defined}\}$	
$ran\, f = \{f(x) \in T : x \in dom\, f\}$	
$f^{-1}(B) = \{x \in dom\, f : f(x) \in B\}$	$(B \subseteq T)$
$A \lhd f : A \rightarrow\!\!\!\!\mid T$	f restricted to A with $dom\ (A \lhd f) = A \cap dom\, f\ \ (A \subseteq S)$

Given $f : S \rightarrow\!\!\!\!\mid T$ and $g : T \rightarrow\!\!\!\!\mid V$,

$g \circ f : S \rightarrow\!\!\!\!\mid V$	composition of f and g with domain $\{x \in dom\, f : f(x) \in dom\, g\}$

Given $f, g : S \rightarrow\!\!\!\!\mid T$,

$$f \oplus g = ((S \setminus dom\, g) \lhd f) \cup g$$

Given functions $f : S \rightarrow\!\!\!\!\mid T$ and $g : S \rightarrow\!\!\!\!\mid U$, then the following schema defines the *product function* $f \times g : S \rightarrow\!\!\!\!\mid T \times U$.

[S, T, U]

$\_ \times \_ : (S \rightarrow\!\!\!\!\mid T) \times (S \rightarrow\!\!\!\!\mid U) \rightarrow (S \rightarrow\!\!\!\!\mid T \times U)$
$f : S \rightarrow\!\!\!\!\mid T,\ g : S \rightarrow\!\!\!\!\mid U$ $f \times g = \{ x \mapsto (f(x), g(x)) \mid x \in dom\, f \cap dom\, g\}$

References

[B84] Brown, N. H., "The EMSP Data Flow Computer", *Proceedings of the 17th Annual Hawaii International Conference on System Sciences*, pp. 39-48, 1984.

[G86] Goguen, J. A., "Reusing and Interconnecting Software Components", *IEEE Computer* **19** (1986), 16-28.

[GRS87] Gerasch, T. E., M. D. Rice, and S. B. Seidman, "Graphical Programming Languages for MIMD Computation", Technical Report TR-10-87, Department of Computer Science, George Mason University, 1987.

[H87] Hayes, I., ed., *Specification Case Studies*, Prentice-Hall, Englewood Cliffs, 1988.

[I88] Inmos Ltd., *occam2 Reference Manual*, Prentice-Hall, Englewood Cliffs, 1988.

[K90] J. Kramer, "Configuration Programming - A Framework for the Development of Distributable Systems", *Proceedings of the IEEE International Conference on Computer Systems and Software Engineering*, Israel, May 1990.

[LF85] LeBlanc, T. J. and S. A. Friedberg, "HPC: A Model of Structure and Change in Distributed Systems", *IEEE Transactions on Computers* **34** (1985), 1114-1129.

[LS90] Lucco, S. and O. Sharp, "Delirium: An Embedding Coordination Language", *Proceedings of Supercomputing '90*, IEEE Computer Society Press, Los Alamitos, 1990, pp. 515-524.

[M89] Milner, R., *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, 1989.

[MKS89] Magee, J., J. Kramer, and S. Sloman, "Constructing Distributed Systems in CONIC", *IEEE Transactions on Software Engineering* **15** (1989), 663-675.

[RY88] Reed, J. and R. T. Yeh , "Specification and Verification of Liveness Properties of Cyclic, Concurrent Processes", *ACM Transactions on Programming Languages and Systems* **10** (1988), 156-177.

[R85] Roscoe, W., "Denotational Semantics of Occam", *Proceedings of the Seminar on Concurrency*, Lecture Notes on Computer Science **197**, Springer-Verlag, Berlin, 1985.

[S88] Spivey, J. M.: *Understanding Z, A Specification Language and its Formal Semantics*. Cambridge University Press, Cambridge, 1988.

[S89] Spivey, J. M.: The Z Notation, A Reference Manual. Prentice-Hall, Englewood Cliffs, 1989.

[SW88] Stovsky, M. P. and B. W. Weide, "Building Interprocess Communication Models using STILE". *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, Vol.2, pp. 639-647, 1988.