AN ALGORITHM TO SUPPORT CODE-SKELETON GENERATION FOR CONCURRENT SYSTEMS*

Maria Heloisa Penedo
Daniel M Berry
Gerald Estrin

Computer Science Department
University of California at Los Angeles

Abstract

Computers are increasingly being used in engineering systems which could utilize a multiplicity of processors. Computer aided design methods are needed to support the design of inherently complex concurrent software. UCLA's SARA (System ARchitects Apprentice) is a design environment which provides computer aid to both hardware and software design of concurrent systems. This paper focusses on an improved capability to aid software design. This capability is provided by defining a Module Interface Description (MID), which allows designers to deal with the structure of code, and effecting a mapping between SARA models and MID-models. It then becomes clear how to create a tool to accept these descriptions and perform various checks on the consistency and completeness of the modeling and implementation descriptions. The addition of this capability reduces the gap between the modeling and realization of systems by providing for automatic generation of code skeletons.

Keywords: computer-aided design, modeling tools, code-skeletons, modules, instantiations.

Introduction

The objective of this paper is to present a *Module Interface Description (MID) Model*, which permits descriptions of the Structure of Code, to be incorporated into the design environment SARA. This MID-model, together with a Model of the Structure of Instantiations is used in the synthesis of concurrent software systems. They support a designer's freedom to express design decisions involving the structure of algorithms. They support automatic checking of inconsistencies between the modeling and implementation and they enable automatic code skeleton generation. In a previous paper [1] the MID-Model was first introduced.

The SARA design environment supports systematic design of hardware and/or software systems. The SARA Design Methodology attempts to provide a designer with effective means for synthesizing and analyzing a system. It is our goal to support a complete path from programming-in-the-large (the act of decomposing a software system into subsystems of modules) [2] to programming-in-the-small (writing code in some algorithmic language) [2]. While existing tools in SARA provide a means for modeling and analyzing a system [3], the MID enhances its power by permitting the expression of design decisions involving the structure of the algorithm. Furthermore the generation of even better code skeletons is possible than with either alone or without the mappings between them.

Section 1 introduces SARA's design process and describes the role that the MID-Model plays in it. In subsequent sections we define the Module Interface Description (MID) Model to be used with the SARA design methodology; we define the relationship between SARA's models and a MID-model; we define the checks that are to be performed to maintain the consistency and completeness of designs which use these models; and we describe an algorithm to generate code skeletons from these models and the mapping between them.

Examples are presented throughout the paper. In order not to get bogged down with detail we have presented our examples with an informality which we believe supports the conceptual level and yet leaves the reader comfortable with the notion that our proposed automation is feasible.

1. The MID role in the SARA System

SARA (System ARchitects' Apprentice)[4, 5, 6] is a computer-aided system which supports a structured multi-level requirement driven methodology for the design of concurrent software or hardware digital systems. The *SARA methodology procedure* is illustrated in figure 1.1a. The *SARA computer-aided system* comprises a number of language processors and tools for assisting the designer using the SARA methodology. Figure 1.1b illustrates the SARA tree, which shows the hierarchy of SARA tools. In this paper we deal with two SARA modeling tools, SL1, which is used to describe structures, and GMB, which is used to describe behavior.

The GMB (Graph Model of Behavior) [7] is a graph-based model for describing behavior in three domains: flow of control, flow of data and interpretation. The primitives used in these three domains are described in Table 1 of the appendix. There is a GMB simulator [8, 9] in SARA, providing an interactive simulation environment which permits experiments on the behavioral models. There is also a Control-Flow Analyzer [10, 11] which allows designers to perform formal analysis on the GMB control graph.

SL1 (Structural Language 1) [12] is a language to describe hierarchically related structures. These structures allow a designer to specify a nested *space of names* which partition a design universe and encapsulate behavioral models. There are three kinds of structural elements: modules, sockets and interconnections. A *module* is used to encapsulate part of a behavioral model. We denote SL1 modules as *s-modules*. A *socket* encapsulates behavior related to the interface between a module and its environment; it is always attached to a module. An *interconnection* connects modules at their sockets; it represents a potential flow of data or control

* This work was supported by the U.S. Department of Energy, Contract No. DE-AS03-76SF00034.

Figure 1.1a UCLA's SARA Design Methodology.

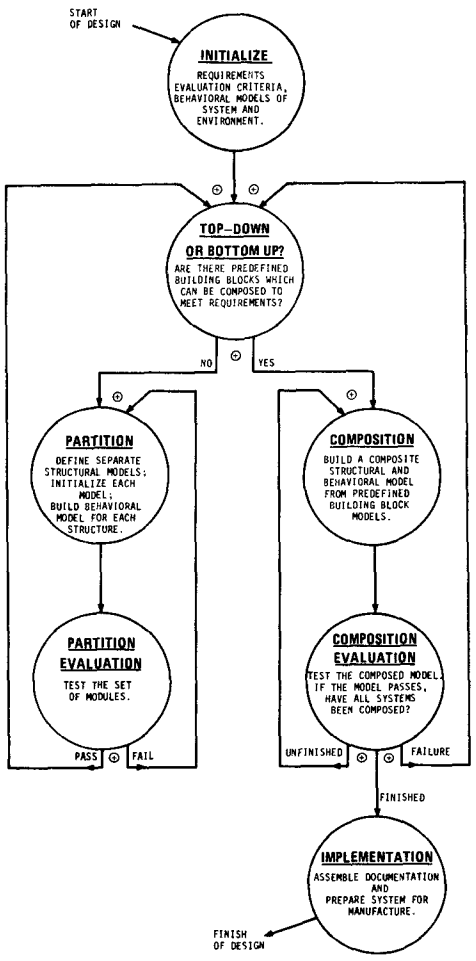
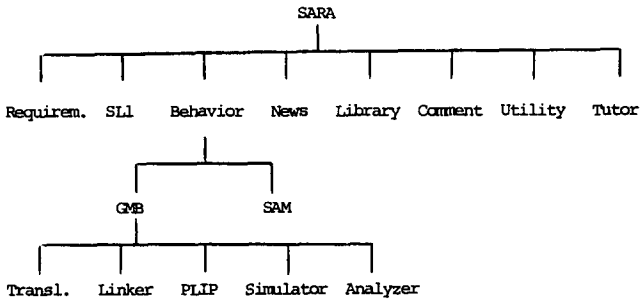


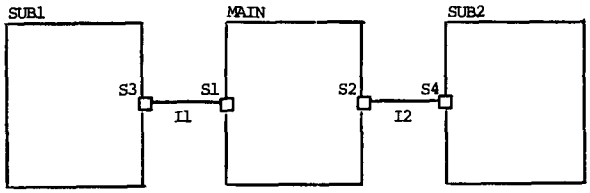
Figure 1.1b SARA tree.



which is made explicit only in its encapsulated behavioral model. Figure 1.2 shows an SL1 model which contains 3 s-modules: MAIN, SUB1 and SUB2. S1 and S2 are sockets representing the interface between module MAIN and its environment. Interconnections I1 and I2 connect MAIN to SUB1 and SUB2.

Since SL1 structures are used to encapsulate behavior, GMB models must be mapped to structures, i.e., GMB subgraphs are mapped to s-modules, GMB arcs crossing boundaries are mapped to sockets and are linked by the corresponding interconnections. Given the mapping between GMB and SL1, and

Figure 1.2 An SL1 model.



given that GMB nodes represent instances, as noted in a previous paper [1], we have identified SL1 as describing the *Structure of Instantiations*. Therefore, throughout this paper we assume that SL1 modules name objects (instances allocated at execution time). We also assume that SL1-sockets encapsulate only instances of procedure calls or instances of procedures being called.

Once a designer has modeled a system by means of the SL1 and GMB models, and analyzed this system by performing analysis or simulation experiments, and once a designer decides to implement this system, then the definition of a MID-model and the mapping between the SL1-model and the MID-model permit selection of a code structure to impose on the system-objects that have been designed. After defining the MID-model and the mapping, if the models pass all checks, the code skeleton can be generated.

2. Module Interface Description Model

A *Module Interface Description Model* is used to describe the structure of code-modules. Its main function is to establish the accessibility of resource names, i.e., procedure names, data type names, etc, among modules, and to assist in binding resource names to modules which provide and need those resources. Other systems and languages such as DREAM [13, 14] and GYPSY [15] include constructs for defining code-module interconnection.

Our proposed MID-model consists of *modules* representing units of code and their *resources* representing interfaces. A MID module is called an *m-module*. An m-module may have hierarchical submodules. An m-submodule represents a unit of code located inside another unit of code that defines its m-module. The name of an m-module is externally visible. Graphically, we represent m-modules by named boxes; these boxes are cut on the upper left corner to distinguish m-modules from SL1-modules. In figure 2.1, *prog, file,* and *sub* are m-modules (units of code). The names *prog, sub* and *file* are visible to each other.

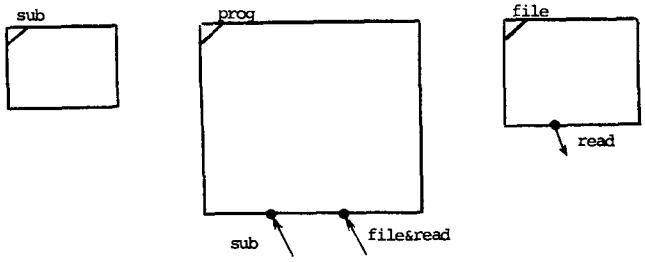


Figure 2.1 A MID model.

Resources can be offered or required. Offered resources define entry points which are externally visible; an m-module name is an implicitly offered resource. Required resources represent names which are required inside the m-module. Graphically, a resource is represented by a small circle attached to an m-module.

If the resource is offered by the m-module, there is a directed arc originating from the resource with an outward direction. If the resource is required by the m-module, there is a directed arc ending at the resource with an inward direction. In figure 2.1, m-module *prog* requires (as indicated by the direction of the arcs) two resources: *sub* and *file$read*. The m-module *file* offers a resource *read*.

In our model, m-modules and resources may have *attributes* associated with them. These attributes are defined by the designer and they are used later in consistency checks and in the generation of code skeletons. In the following subsections we describe a set of attributes that can be associated with m-modules and resources. The values of the *genre attribute* (both for m-modules and resources) were designed with a particular *modeling scheme* in mind, in which concurrent processes communicate through shared data via procedure calls, and all the synchronization is associated with the shared data objects. We envision extensions of this particular set of attributes to provide for other schemes, such as message passing, etc.

2.1 Proposed m-Module Attributes

A module, in a MID-model, represents the *name of a unit of code*, such as a *procedure, a data type definition.*

The *attributes* of an m-module are:

1) *Name* - an identifier which is the external name which identifies that unit of code.

2) *Synonym* - an optional identifier which is a synonym for the name of an m-module, i.e., it is also an external name identifying the unit of code.

3) *Specification* - of the behavior of this unit of code, written in some specification language such as first-order predicate calculus (This paper does not deal with this part.).

4) *Genre* - the genre of an m-module tells the nature of the unit of code. We have defined four genres:

i) *Process definition.* It represents a piece of code which defines a process type, similar to the process type in concurrent Pascal [16]. Variables can be declared of type *process;* they represent instances of this particular type. A process cannot be called; an instance of a process is initialized upon initiation of the system of which it is a part. Many instances of processes may execute concurrently.

ii) *Shared data type* (sdt). It represents a unit of code which implements a shared data type. An instance of this type can be a shared data object, i.e., a data object which can be accessed concurrently by more than one process instance. It consists of the representation of the data, the operations which access this data, and an implied *synchronization* mechanism. An instance of this type consists of a data cell with type as specified by the representation, instances of the operations, and the implied synchronization mechanism for this instance.

iii) *Abstract data type* (adt). It represents a piece of code which implements an abstract data type, i.e., the representation of the data and the operations to access the data, or a group of procedure definitions. An instance of this m-module cannot be shared, i.e., it cannot be accessed concurrently by more than one process instance.

iv) *Procedure Definition.* It represents a piece of sequential code which implements an algorithm. A procedure can be called, i.e, it can be instantiated during the execution of the system of which it is a part.

2.2 Resource Attributes

A resource, in a MID-model, represents the *name of a procedure,* the *name of a data type,* or the *name of a data type operation,* which is offered or required by the m-module to which this resource is attached. The name of an m-module, with the exception of genre *process,* is considered an implicitly offered resource.

The *attributes* of a resource are:

1) *Name* - an identifier defining the external name of the required or offered resource.

2) *Synonym* - an optional identifier which is a synonym for the name of the resource.

3) *Genre* - the genre of a resource can be: procedure, entry point, data type or data type operation.

4) *Parameter types* - if any, with an order implied.

5) *Return value type,* if any.

6) *Specification* - of the behavior of the interface. (The authors will not deal with this part.)

7) *Resource Owner and Owner Genre* - m-module *name* and *genre* of the owner (the one which offers this resource) of the resource, in the case that the resource is required and the owner is not the resource itself.

2.3 Examples of MID-models

For the MID-model illustrated in figure 2.1, we could define the following attributes:

```
MOD PROCESS prog,
  SYNONYM program,
  REQUIRES
    PROC sub (int, int, int) bool,
    PROC read (int) char, OWNER SDT file;
END_MOD;

MOD PROC sub (int, int, int) bool;
END_MOD;

MOD SDT file,
  OFFERS
    PROC read (int) char;
END_MOD;
```

There are three m-modules with names *prog, file* and *sub.* The m-module *prog* has a SYNONYM *program,* and GENRE *process;* it requires a resource *sub* of GENRE *procedure* with three parameters of type integer and a return value of type boolean, and it requires a resource *file* of GENRE *sdt* with operation *read.* The m-module *file* has GENRE *sdt,* and it offers resource *read* with a parameter of type integer and a return value of type character.

Figure 2.2a shows an m-module *trans* with m-submodules *order* and *stack*. Let us say that m-module *order* has GENRE procedure and m-module *stack* has GENRE adt with resources *insert* and *remove*. This figure implies that the code definition for procedure *order* and adt *stack* are internal to the code for *trans*, and thus are not accessible outside *trans* unless *trans* has a resource offering the procedure name *order* or the data type definition *stack*. On the other hand, figure 2.2b shows an m-module *trans* with m-submodule *order*. The *stack* m-module is external to *trans* implying that the code for *stack* is external to the code for *trans*, and thus is accessible to other units at the same level.

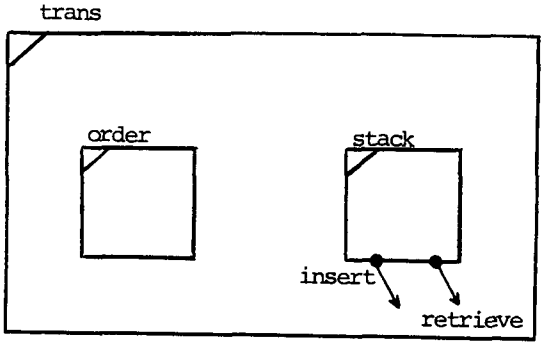


Figure 2.2a Example of a MID model.

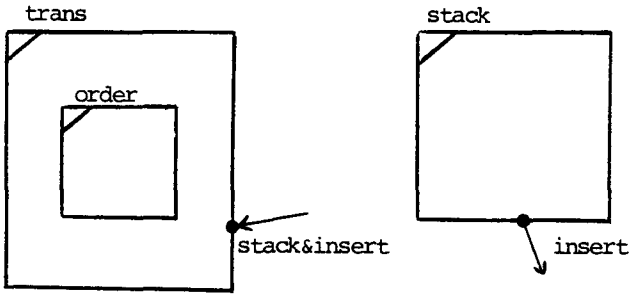


Figure 2.2b Example of a MID model.

3. Association between an SL1-GMB and a MID-model

Given that an SL1-GMB model represents a *structure of instantiations* [1] and a MID-model represents a *structure of algorithms* the mapping between these models should express the designer's decisions as to which SL1-GMB modules are instances of which m-modules, i.e., which data objects are instances of what data types, which process and procedure instantiations are instances of which process and procedure definitions.

The *mapping between SL1 and MID* is defined as follows:

1. every s-module must be mapped to a unique m-module,

2. every socket in an s-module A must be mapped to a resource, either offered or required, of an m-module *a*. Note that s-module A must be mapped to this m-module *a*. Note also that a socket may be mapped to an m-module itself, with certain restrictions, since an m-module is implicitly an offered resource.

The relationship between s-modules and m-modules can be one-to-one, many-to-one or even zero-to-one. It is important to note that, when a system being modeled happens to have instantiations one-to-one with code [5] then one can get a code skeleton from the SL1-GMB models but this process does not generalize.

Let us suppose that the SL1-GMB model shown in figure 3.1 has been designed and analyzed. Let us also suppose that the designer, by looking at the behavior encapsulated by s-modules SUB1 and SUB2, decides that they can both be implemented by the same procedure definition with their differences parameterized. Figure 3.2a shows the result of this design decision in the definition of the illustrated MID-model and the mapping between the SL1-GMB model of figure 3.1 and this MID-model. Thus both s-modules, SUB1 and SUB2, are defined, by the mapping, as *instances*

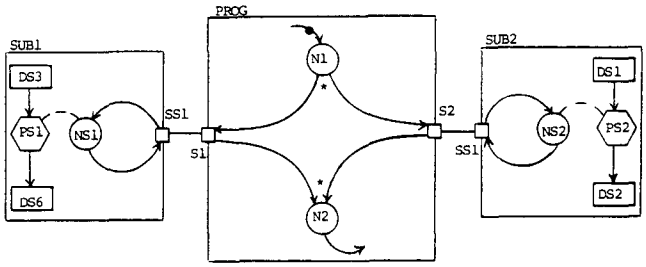
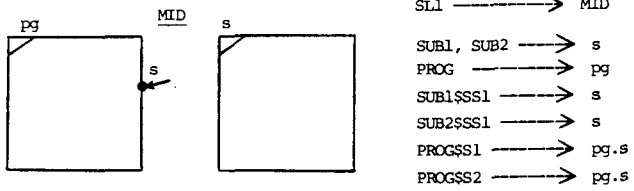


Figure 3.1 An SL1-GMB model.

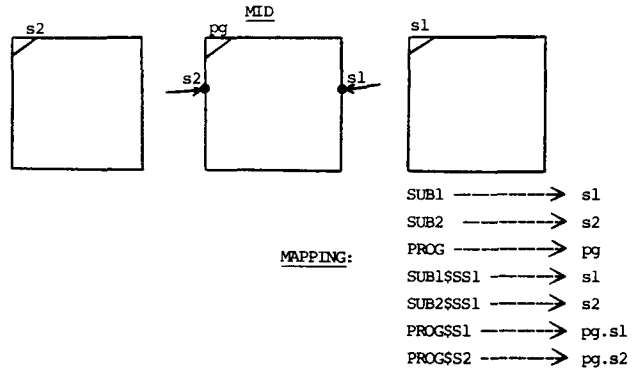
Figure 3.2a A Design Decision.



of m-module *s*. Sockets SS1, in both SUB1 and SUB2, are mapped to resource m-module *s;* sockets S1 and S2 in PROG are mapped to resource *s* in *pg*.

Alternatively, if the designer decides to implement SUB1 and SUB2 by two different procedure definitions, then the MID-model and mapping are defined as illustrated in figure 3.2b.

Figure 3.2b An alternative design decision.



From now on, whenever we refer to the *m-module corresponding to an s-module*, we mean the m-module to which the s-module is mapped.

4. Checks

In this section we list some of the checks to be applied to a MID-model and to the mapping between SL1-GMB and MID. They are intended to check the consistency and completeness of the MID-model and the mapping between this model and the structure of the instantiations. Some of these checks may be done at the time the designer is defining the models and/or at designer's request. It is required that a design pass these checks *before* the code skeleton is generated.

MID checks

- Let the name list of an m-module be the list consisting of its name and its synonyms, if any. Then all name lists must be unique.

- All required resources exist as offered resources within their visibility space, and the attributes of each pair (offered, required) are compatible. In the multilevel case, if an m-module *a* requires a resource which is not offered within that level, but is offered at the same level as the m-module which surrounds *a*, then m-module *a* must also require that resource.

- Process names are not offered as resources since they cannot be called. Note that there is no resource genre *process*.

- If an m-module is of genre *procedure* or *adt*, its required resources can be resources only of genre *procedure* or *adt*.

SL1-MID MAPPING checks

- Each s-module M is mapped to one and only one m-module *m*, i.e., M is an instance of *m*.

- If an instance of m-module *p* has as submodule an instance of m-module *q* (in the SL1-model), then *p* must have access to *q*.

- Each socket *s* is mapped to one resource *r*. If socket *s* belongs to s-module *A*, which is mapped to m-module *a*, then the resource *r* must be a resource of *a* (if it is not already m-module *a* itself).

- If sockets *sa* and *sb* are externally connected and they are mapped to resources *pa* and *pb*, as illustrated in figure 4.1, then

  a) *pa* and *pb* must have identical names, or they must be synonyms.

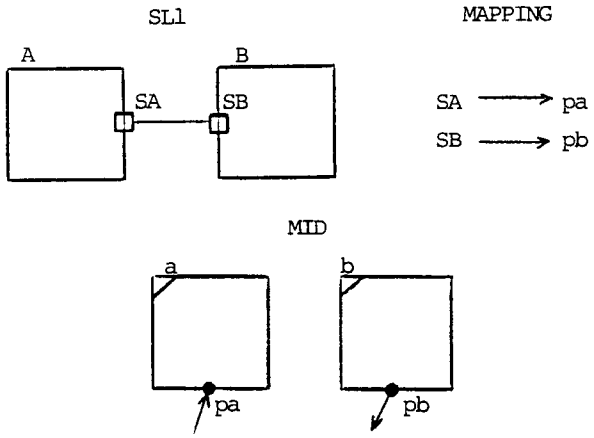
  b) *pa* and *pb* must have the same number of parameters, the same types of parameters and return value.

  c) if *pa* (or *pb*) is a required resource, the *owner* of this resource *pa* (or *pb*) must have the same name (or be a synonym) and the same genre as the m-module which has the resource *pb* (or *pa*).

- For each m-module *a* (if there is an s-module *A* mapped to it), for each required resource *i* in *a*, as illustrated in figure 4.2, there is either:

  a) a socket in *A*, mapped to this resource, (which represents an instance of an operation call), as illustrated in figure 4.2a, or

Figure 4.1 Checks - Example 1.

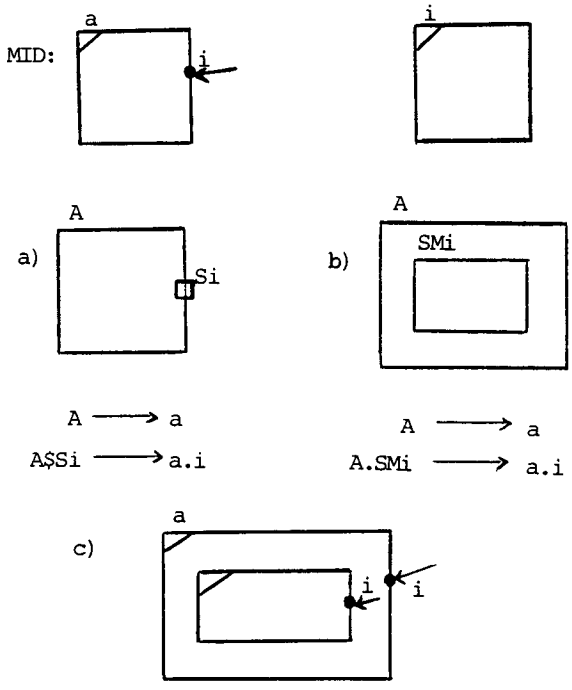


  b) an s-module *SMi* (submodule of *A*) mapped to the m-module which corresponds to the required resource, as illustrated in figure 4.2b. (This s-module *SMi* represents an instance of a procedure or a data type), or

  c) an internal required resource, which has no matching internal offered resource (figure 4.2c).

- If an instantiation of an m-module of genre *adt* is required by more than one process instantiation (i.e., the data may be shared) then a warning is given since there is no inherent synchronization mechanism.

Figure 4.2 Checks - Example 2.



5. Generation of code skeletons

In this section we specify a way to generate code skeletons from an SL1-GMB model, a MID-model, and the mapping between the two. First we specify the basic rules and second we describe the algorithm to automatically generate the code skeleton.

General Rules

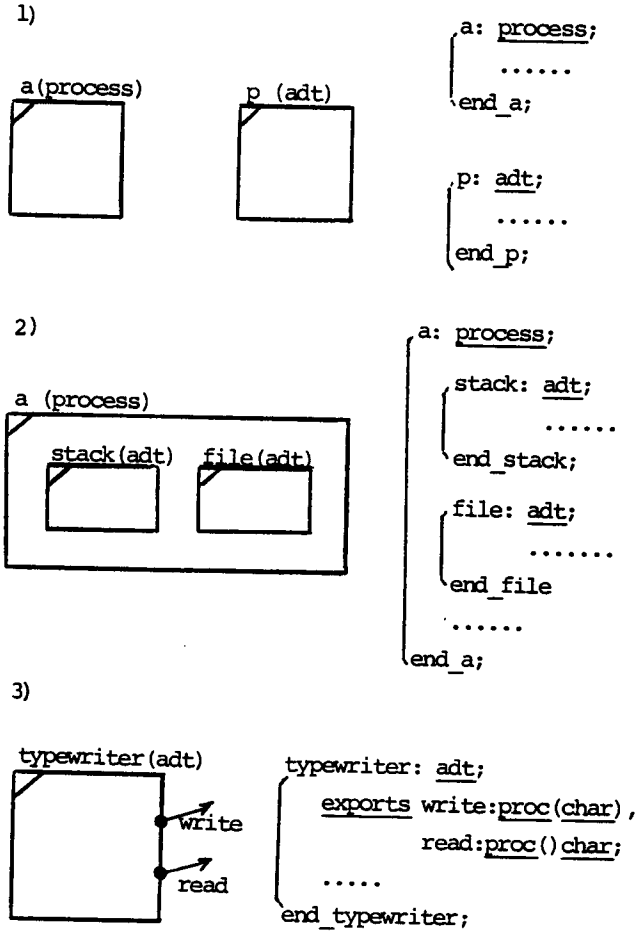
Briefly, the SL1-GMB model identifies the *variables* (which name objects) and the *external objects* needed inside each module. The MID-Model identifies the *types* of the objects whose definitions correspond to units of code. The mapping says *which variable is of what type* and *which external object is needed by what m-module.* The nested definition of MID modules determines the nesting of the code and the nested definition of SL1 modules determine the location of the objects' declarations.

The rules for the *generation of code skeletons,* from SL1-GMB and MID-models, are as follows:

From the MID-model

1) Each m-module corresponds to a unit of code (code-module) with name and type as specified in the MID attributes for the m-module (illustrated in figure 5.1.1).

Figure 5.1 Examples of Code-Skeleton Generation.

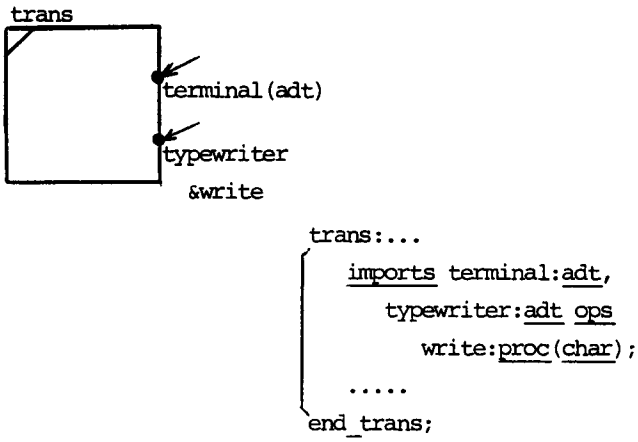


2) If an m-module *a1* is a submodule of m-module *a,* then the unit of code for *a1* will be located internal to the unit of code for *a.* In figure 5.1.2, the units of code for m-modules *stack* and *file* are located internal to the unit of code for *a.*

3) For each offered resource in an m-module there is a specification of this resource as being *exported,* be it of genre *adt, procedure,* or *operation,* as illustrated in figure 5.1.3. The type of the parameters and return value are as specified in the attributes for this resource.

4) For each required resource in an m-module, there is a declaration of this resource as an *imported* object, with the type as specified in the attributes for this resource, as illustrated in figure 5.1.4.

4) Figure 5.1 continued.



From SL1 and Mapping

5) Each s-module corresponds to a variable or an instance of a call, with name as defined in the SL1 model, as illustrated in figure 5.1.5. The type of this variable or call is the name of the m-module to which this s-module is mapped.

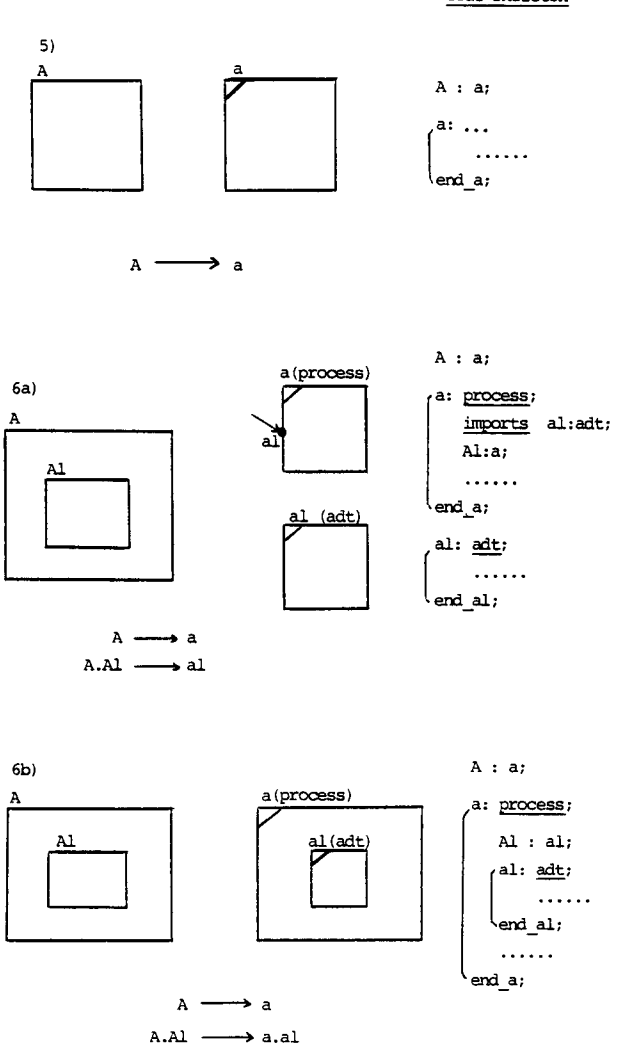
If the genre of the m-module *a* is *adt,* or *sdt* then *A* names an instance of this data type definition. If the genre of the m-module *a* is *process* then *A* names an instance of this process definition, which is allocated at initiation of the execution of the system. If the genre of the m-module *a* is *procedure,* then *A* names an instance of this procedure, which is instantiated upon call, very similar to a value of type *TASK* in PL1. Depending on the target programming language, either a declaration of the above form **A : a** is generated, or we use this name to label a call to the procedure definition, i.e.,

A : **call** a (<list_of_parameters>).

6) If an s-module *A1* is a submodule of an s-module *A* (which is mapped to the m-module *a),* then the variable corresponding to *A1* is declared internal to the unit of code for *a.* Figures 5.1.6a and 5.1.6b illustrate this case, when submodule *A1* is mapped to m-module *a1* and *a1* is defined external (figure 5.1.6a) and internal (figure 5.1.6b) to m-module *a.* Note that in the first case there is a declaration of the imported object *a1 : adt,* since *a1* is a required resource, as explained in item 4 above.

7) If there are two or more s-modules *Ai(i=1,n),* which contain submodules, mapped to the same m-module *a,* then for each such s-module *Ai* mapped to the same m-module, there is a set *Si(i=1,n)* of variables corresponding to the submodules, as

Figure 5.1. (Cont.)



illustrated in figure 5.1.7. The sets *Si(i=1,n)* correspond to one set *S* (of variables) in the code; the names in one set may be used or all sets may be defined as 'aliases' in the code. In the figure, *B1* is declared an alias of *B2*.

8) For every output socket in an s-module (which represents a call to a procedure) there is a specification of the external object it accesses as being *imported,* together with its proper type. The name of the external object is obtained from the first name in the full path name of the first input socket to which the output socket is connected; it is in fact the name of the module which has this input socket. The type is obtained from the s-module's corresponding m-module.

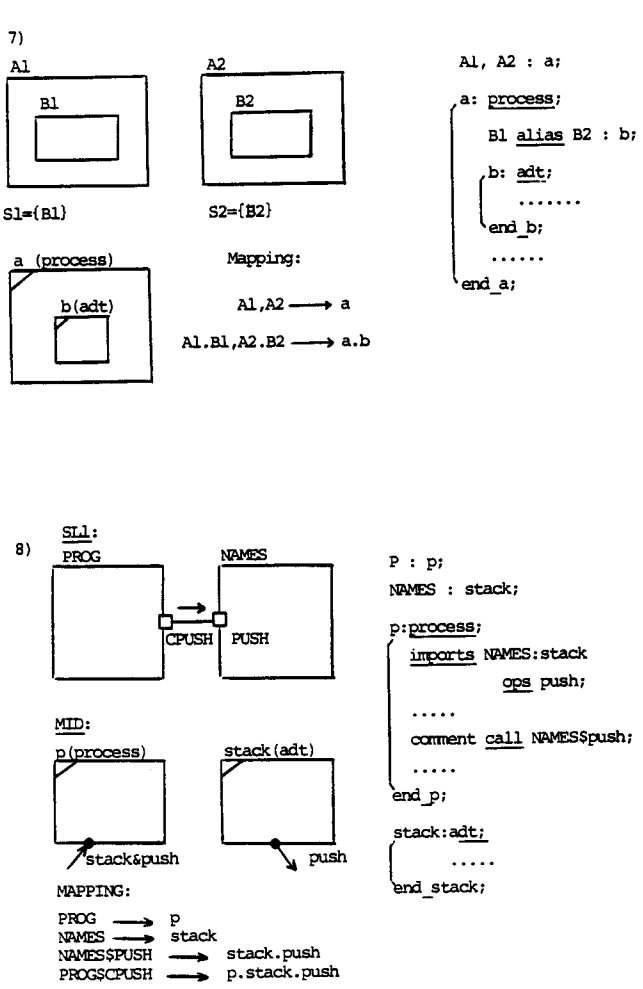
There also is a corresponding call to the named resource; the instance of that type is the name of the s-module.

In figure 5.1.8, output socket cpush generates the declaration

imports NAMES : stack
<comment **call** NAMES$push()>

NAMES was obtained from the first name in the socket's full name, the type was obtained from the mapping (NAMES==>stack), and the operation is the name of the resource.

Figure 5.1 continued.



5.1 Algorithm

We now specify the *algorithm* which generates a code skeleton.

Briefly it consists of two steps: Step 1 translates the SL1-model, i.e., it generates declarations of the objects with their associated types. The location of these declarations is determined by their nesting position in the SL1-model. It also generates the list of external objects which are imported by each m-module. Step 2 transforms the MID-model into units of code and generates the exported and imported type definitions.

To maintain the uniformity of the algorithm, we add an m-module, called CODE, surrounding the MID-model; at the end of the algorithm the complete code skeleton is associated with this m-module CODE.

Algorithm

STEP 0 - Initialize

a) Surround the MID-model with an m-module called CODE.

b) Mark all SL1-modules as 'non-translated'.

c) Mark all MID-modules as 'non-translated'.

STEP 1 - Translate SL1

a) Select a non-translated SL1-module with no submodules, or an SL1-module all of whose submodules are marked 'translated'. Let us call the selected s-module, *A*.
If all SL1-modules are marked translated, this step is complete; go to Step2.

b) Translate the selected s-module A and its sockets into variables and declarations:

i) Generate the declaration,
A : <name of A's corresponding m-module>

If there is a synchronization specification associated with this s-module, add to the declaration above,
with <synchronization_spec>

Associate (add) this declaration with the code for the *location_module*. The location_module is: s-module CODE, if A is the outermost s-module; otherwise, obtain it in the following way:
- obtain the name of the s-module that surrounds A; call it *father_of_A*.
- obtain the father_of_A's corresponding m-module; call it *location_module*.

ii) For every output socket in s-module A, generate a declaration of the following form:
imports <object_name> : <type>
where these names are obtained as follows:
- get the first input socket connected to this output socket; call it the *end_socket;*
- obtain the name of the s-module to which the end_socket is attached; call it the *end_module_name;*
- get the name of the end_socket's corresponding resource; call it *res_name;*
Then,
<object_name> is the end_module_name,
<type> is the name of the end_module's corresponding m-module.

Also generate a comment of the following form:
<comment **call** <object_name>()>, if the type is the same as the res_name; or
<comment **call** <object_name>$<res_name>()>,
if the type is different than the res_name.

Associate this code generated with A's corresponding m-module.

c) Mark this s-module as 'translated'.

d) Go to 1a.

STEP 2 - Translate MID:

a) Select a non-translated MID-module with no submodules or a MID-module all of whose submodules are marked 'translated'. Let us call this selected m-module *m*.
If all m-modules are marked translated, then this step and the algorithm are complete. The outermost module, CODE (added in the initial step), has a complete code skeleton associated with it.

b) Translate this m-module *m* and its resources into units of code and declarations of imported and exported resources:

i) For each *offered resource* in *m*, generate:
exports <res_name>:<res_genre><res_attrs>,
if *m* is the same as the resource_owner; or
exports <res_owner>:<res_genre>
ops <res_name>:<res_genre><res_attrs>,
if *m* is not the resource owner;
where <res_attrs> are the type of the parameters and return value defined for the resource.

ii) For each *required resource* in *m*, generate
imports <res_name>:<res_genre><res_attrs>,
if the resource name is the same as resource_owner; or
imports <res_owner>:<res_owner_genre>
ops <res_name>:<res_genre><res_attrs>,
if the resource_name is not the same as the resource_owner.

iii) Add the declarations generated in items i and ii to the code associated with *m*.

iv) Generate a code-module of the form
m : <genre of m>
end_m;
Fill in this code-module with the code associated with *m*, and add this code to the code associated with the m-module which surrounds *m*.

c) Mark this m-module as 'translated'.

d) Go to 2a.

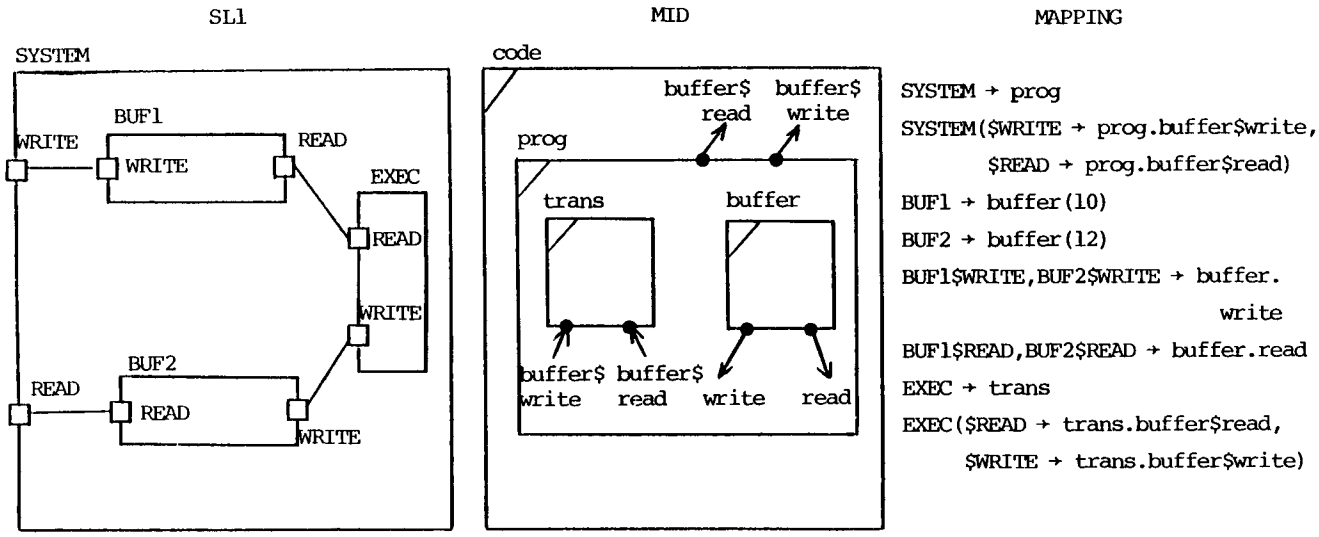
5.2 Example

We now give an example of the application of this algorithm. Let us assume that the SL1 model illustrated in figure 5.2 represents the structure of a system which has been previously modeled (by the use of GMBs) and analyzed. The GMB and associated interpretation are not shown in the example. Let us also assume that the MID-model and the mapping, illustrated in the same figure (figure 5.2), have been defined by the designer, and all checks have been passed. We can then perform the Code Skeleton Generation algorithm. The mapping indicates that SYSTEM is an instance of *prog*, BUF1 and BUF2 are instances of *buffer* of size 10 and 12 respectively, and EXEC is an instance of *trans*. Let us assume that m-modules *trans* and *prog* are defined with genre *process*, and m-module *buffer* is defined with genre *sdt*. In the figure we already show the m-module CODE surrounding the MID-model.

At the end of *Step 1* (Translate SL1), the following translation has ocurred:

a. associated with m-module *code:*
SYSTEM : prog;

Figure 5.2 An Example.



Note: EXEC$READ and EXEC$WRITE are output sockets.

b. associated with m-module *prog:*

```
BUF1 : buffer(10) with sync <sync_spec>
BUF2 : buffer(12) with sync <sync_spec>
EXEC : trans
```

c. associated with m-module *trans,*

```
imports BUF1 : buffer
<comment call BUF1$read()>
imports BUF2 : buffer
<comment call BUF2$write()>
```

At the end of *Step 2,* the following complete code skeleton is associated with m-module CODE:

```
SYSTEM : prog;
prog:process;
    exports buffer:sdt ops write:proc(char(80)),
          buffer:sdt ops read:proc()char(80);
    BUF1: buffer(10) with sync <sync_spec>;
    BUF2: buffer(12) with sync <sync_spec>;
    EXEC: trans;

    trans : process;
          imports buffer:sdt
                ops write:proc(char(80)),
               buffer:sdt
                ops read:proc()char(80);
          imports BUF1 : buffer,
                BUF2 : buffer;

          <trans_body, which includes:>
            <comment call BUF1$read()>
            <comment call BUF2$write()>
    end_trans;

    buffer : sdt(n);
          exports write:proc(char(80)),
                read:proc() char(80);
           <buffer_body>
    end_buffer;

    <prog_body>
end_prog;
```

Note that we have used <trans_body>, <buffer_body>, etc, to represent the actual code for each unit, which is not generated by the algorithm. If, in the design process, the designer defines the code to be associated with each m-module, then this code may be used to fill in the code skeletons.

6. Conclusions

In a previous paper [1], we have established the need for the distinction between the structure of instantiations and the structure of code to provide for parallel processing and multiple declarations of data objects of the same type.

In this paper, we have described a MID-Model to permit expressions of decisions involving the structure of code. We have shown how this MID Model can enhance SARA by facilitating the progression from modeling to code. We have listed some of the checks to be performed during this transition, and we have described an algorithm which supports automatic generation of code skeletons.

We have defined a small set of attributes to be associated with the MID primitives; we are encouraged to search for other attributes in order to support other schemes.

Further research and development is required to realize the tool to support this MID-model. This research involves the investigation of a data structure for storing the MID-models, ways to implement the mapping between SL1-GMB and MID, and ways to implement the checks. There is a significant amount of information that the designer must contribute to support these models, and we are not yet satisfied that we have minimized that burden. However the automatic checking that can provide for early detection of design and implementation inconsistencies and incompleteness does justify a burden on the designer. Design checking should produce more reliable software. Furthermore the history of the design is documented, i.e., stored, providing for better maintenance of designs. We do not know of any other systems which support this kind of design more effectively at a *lower burden.*

We have discussed the transition between modeling and code for programming-in-the-large. We have not discussed

programming-in-the-small, the translation between the models and the statements of the implementation language. One of the authors of this paper, in her Ph.D. dissertation [17], has probed some very important issues dealing with the transition from GMB-models into code (programming-in-the-small). That contribution encompasses:

- The analysis of the GMB model as used to model abstractions of procedures and data, instantiations, and synchronization mechanisms.
- The proposal of high-level primitives to model procedure calls.
- The definition of a data abstraction mechanism in SL1-GMB.
- The definition of a mapping between the synchronization mechanism Predicate Path Expressions [18], and GMB.
- The proposal of a Structured SL1-GMB, which consists of structured GMB constructs, together with design guidelines. This proposal leads to structured and checkable designs and, moreover, these models can be taken all the way to code.
- The definition of a subset of an Interpretation Language to be used in the SARA methodology which contains constructs which map to the Structured SL1-GMB.
- The definition of an abstraction algorithm to take models defined using the Structured SL1-GMB, to code, written in the proposed interpretation language. The path from this interpretation language to any of a set of existing programming languages is claimed to be simple.
- The importance of the MID-model and mapping to development of software building-blocks.
- The specification of new tools to be integrated in SARA to support all of the above.

References

1 Penedo, M. H. and D.M. Berry, "The Use of a Module Interconnection Language in the SARA System Design Methodology", *Proceedings of the 4th International Conference on Software Engineering*, Munich, Germany (September, 1979).

2 DeRemer, F. and H.H. Kron, "Programming-in-the-Large versus Programming-in-the-Small", *IEEE Transactions on Software Engineering*, Vol. 2, No. 2, pp. 80-86 (June 1976).

3 Razouk, R.R., M. Vernon, and G. Estrin, "Evaluation Methods in SARA-the Graph Model Simulator", *Proceedings of the Conference Simulation, Measurement and Modeling of Computer Systems*, Boulder, Colorado (August 1979).

4 Estrin, Gerald, "A Methodology for Design of Digital Systems - supported by SARA at the Age of One", *AFIPS Conference Proceedings*, Vol. 47 (1978).

5 Campos, I.M. and G. Estrin, "Concurrent Software System Design Supported by SARA at the Age of One", *Proceedings of the 3rd International Conference on Software Engineering*, pp. 230,242 (1978).

6 Campos, I.M. and G. Estrin, "SARA aided design of Software for concurrent systems", *AFIPS Conference Proceedings*, Vol. 47 (1978).

7 Gardner, R., W. Overman and W. Ruggiero, "GMB System Reference Manual", Computer Science Department, UCLA (July 1977).

8 Razouk, R.R. and G. Estrin, "The Graph Model of Behavior Simulator", *Proceedings of the Symposium on Design Automation and Microprocessors*, pp.67-76 (February, 1977).

9 Razouk, R.R., "GMB Simulator System Reference Manual", Computer Science Department, UCLA (1979).

10 Razouk, R.R., "Computer Aided Design and Evaluation of Digital Computer Systems", Ph.D. Dissertation, Computer Science Department, UCLA (1980).

11 Razouk, R., M. Vernon, and M. Brewer, "Control-Flow Analyzer User Manual", Computer Science Department, University of California, Los Angeles (February 1980).

12 Penedo, M.H., "SL1 System Reference Manual", Computer Science Department, University of California, Los Angeles, CA (February 1979).

13 Riddle, W.E. et al, "An Introduction to the DREAM Software Design System", *Software Engineering Notes*, Vol.2, No. 4, pp. 11-23 (July 1977).

14 Riddle, W.E. et al, "Behavior Modeling during Software Design", *IEEE Transactions on Software Engineering*, SE-4, No. 4 (July 1978).

15 Report on the Language GYPSY - Version 2.0, ICSCA-CPM-10, Institute for Computing Science and Computer Application, University of Texas, Austin, Texas (May 1978).

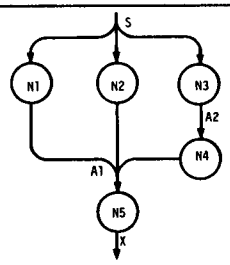
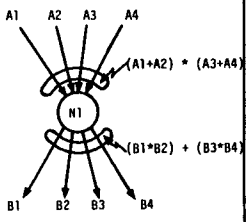
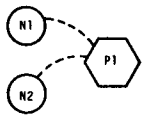
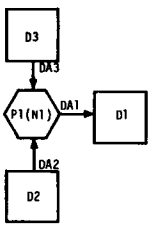
16 Brinch Hansen, P., *Operating Systems Principles*, Prentice-Hall, Englewood Clifs, N.J. (1973).

17 Penedo, M.H., "The Use of a Module Interface Description in the Synthesis of Reliable Software Systems", Ph.D. Dissertation, Computer Science Department, UCLA (1980).

18 Andler, S., "Predicate Path Expressions: A High Level Synchronization Mechanism", Ph.D. Dissertation, Department of Computer Science, CMU (August 1979).

Appendix

Table 1. GMB control and data primitives.

BEHAVIORAL PRIMITIVES TYPE	GRAPHICAL	MACHINE PROCESSABLE
A NAMED CONTROL NODE REPRESENTS A STEP IN A PROCESS BEING MODELED. A CONTROLLED DATA PROCESSOR (SEE BELOW) MAY BE ASSOCIATED WITH A NODE TO PROVIDE INTERPRETATION OF THE PROCESS. EXAMPLE: A NODE N1 HAS A SINGLE ENTRY ARC S AND A SINGLE EXIT ARC X.		@CONTROL_GRAPH; @NODES N1; @ARCS S,X; N1 (S:X); @END;
A NAMED DIRECTED CONTROL ARC REPRESENTS NON-VOLATILE PRECEDENCE RELATIONS BETWEEN SETS OF NODES. IF THERE IS MORE THAN ONE SOURCE OR DESTINATION NODE THE ARC IS CALLED COMPLEX; OTHERWISE IT IS CALLED SIMPLE. AN ENABLING TOKEN IS PLACED ON AN ARC EITHER AS A STARTING STATE OR UPON TERMINATION OF ANY OF ITS SOURCE NODES. WHEN A NODE IS INITIATED, ITS ENABLING TOKENS ARE ABSORBED. EXAMPLE: A2 AND X ARE SIMPLE CONTROL ARCS. A1 IS A COMPLEX CONTROL ARC WHOSE SOURCE SET IS NODES N1, N2 and N4 AND WHOSE DESTINATION SET IS N5. S IS AN INCOMING COMPLEX ARC WHOSE DESTINATION SET IS N1, N2, AND N3. IF THERE WERE AN INITIAL TOKEN ON S, THE TOKEN MACHINE MECHANISM WOULD NON-DETERMINISTICALLY ENABLE N1 OR N2 OR N3 AND THE TOKEN WOULD BE ABSORBED.		@CONTROL_GRAPH: @NODES N1,N2,N3,N4,N5; @ARCS S,A1,A2,X; N1 (S:A1); N2 (S:A1); N3 (S:A2); N4 (A2:A1); N5 (A1:X); @END;
INPUT CONTROL LOGIC A LOGICAL RELATION AMONG THE INPUT ARCS TO A NODE SPECIFIES THE PRECEDENCE CONDITIONS THAT MUST BE SATISFIED BY TOKEN STATES FOR THE NODE TO BE INITIATED. TOKENS FROM THE INITIATING ARCS WHICH SATISFY THE INPUT RELATIONS ARE ABSORBED BY THE TOKEN MACHINE. TOKENS ARE ABSORBED FROM ONE OF AN INITIATING ARC SET GOVERNED BY AN OR RELATION IN A MANNER ESTABLISHED IN THE TOKEN MACHINE AND FROM ALL MEMBERS OF AN INITIATING ARC SET GOVERNED BY AN AND RELATION. EXAMPLE: IF ENABLING TOKENS EXIST ON EITHER A1 OR A2 AND ON EITHER A3 OR A4 THEN N1 CAN BE INITIATED. OUTPUT CONTROL LOGIC A LOGICAL RELATION AMONG THE OUTPUT ARCS SPECIFIES WHICH ARCS HAVE TOKENS PLACED UPON THEM WHEN A CONTROL NODE IS TERMINATED. WHEN AN EXCLUSIVE OR OUTPUT RELATION HOLDS, A DATA PROCESSOR INTERPRETATION MUST DECIDE WHICH ARC RECEIVES A TOKEN. WHEN AN AND RELATION HOLDS ALL OUTPUT ARCS RECEIVE TOKENS. EXAMPLE: WHEN N1 TERMINATES, ITS ASSOCIATED CONTROLLED DATA PROCESSOR WILL HAVE DECIDED WHETHER TOKENS ARE TO BE PLACED ON B1 AND B2 OR ON B3 AND B4.		INPUT: OUTPUT CONTROL LOGIC @CONTROL GRAPHS; @NODES N1; @ARCS A1,A2,A3,A4,B1,B2,B3,B4; N1((A1+A2) * (A3+A4); (B1+B2) * (B3+B4)); @END;
A NAMED CONTROLLED DATA PROCESSOR REPRESENTS A DATA TRANSFORMATION OBJECT WHICH IS ACTIVATED WHEN AN ASSOCIATED CONTROL NODE IS INITIATED. E.G., PROCESSOR P1 IS INITIATED WHENEVER EITHER N1 OR N2 IS INITIATED. WHEN PROCESSOR P1 TERMINATES IT CAUSES TOKENS TO BE PLACED ON OUTPUT ARCS OF THE CONTROL NODE WHICH INITIATED IT. AN INTERPRETATION OF THE DATA TRANSFORMATION AND OTHER PARAMETERS SUCH AS TIME DELAY OR RESOURCE REQUIREMENTS CAN BE ASSOCIATED WITH THE DATA PROCESSOR. EXAMPLE: PROCESSOR P1 HAS A RANDOM DELAY ASSOCIATED WITH IT. IRAND IS A BUILT-IN FUNCTION. THE CONTROL GRAPH CARRIES THE BURDEN OF GUARANTEEING THAT N1 AND N2 ARE ENABLED IN A DESIRED SEQUENCE. OTHERWISE THEY WILL BE ACTIVATED IN A NON-DETERMINISTIC ORDER AND THE SIMULATOR WILL SHOW POSSIBLE CONTENTION.		@DATA GRAPH; @PROCESSOR P1 (N1,N2); @END; PLIP INTERPRETATION @PROCESSOR P1; DCL IRAND ENTRY(FIXED BIN(31) FIXED BIN(31))RETURNS (FIXED BIN(31)); /*RANDOM # GENERATOR*/ DCL NUMBER FIXED BIN(31); NUMBER=IRAND(1,2) /*PICK AN INTEGER: 1 OR 2*/ IF NUMBER=1 THEN @OUTPUT_ARCS='B1,B2'; ELSE @OUTPUT_ARCS='B3,B4'; @DELAY=IRAND (10,100); /*PICK RANDOM DELAY FROM 10 TO 100*/ @END PROCESSOR;
A NAMED UNCONTROLLED DATA PROCESSOR REPRESENTS A DATA TRANSFORMER WHICH PROVIDES, AT ITS OUTPUT, STATED FUNCTIONS OF ITS INPUTS INDEPENDENT OF CONTROL NODE STATES. IN THE DATA GRAPH AN UNCONTROLLED PROCESSOR IS IDENTIFIED BY PROVIDING AN EXPLICIT DECLARATION. AN INTERPRETATION OF THE DATA TRANSFORMATIONS AND OTHER PARAMETERS MAY BE ASSOCIATED WITH IT IN AN IDENTICAL MANNER TO THE CONTROLLED PROCESSOR.		@DATA_GRAPH; @UNCONTROLLED_PROCESSORS U1; @END;
A NAMED DATA SET REPRESENTS A PASSIVE COLLECTION OF DATA. DATA STRUCTURE MAY BE ASSOCIATED WITH A DATASET. ALL PL/1 DECLARATIONS NOT CONTAINING SCOPE OR STORAGE CLASS ATTRIBUTES ARE ACCEPTED AS DEFINITIONS OF DATA SETS. CHARACTER STRINGS CANNOT HAVE THE VARYING ATTRIBUTE. EXAMPLE: THE DATASET D1 IS A SIX-DECIMAL-DIGIT COMPLEX FLOATING POINT NUMBER.		@DATA_GRAPH; @DATASETS D1; @END; PLIP INTERPRETATION @DATASET D1 COMPLEX FLOAT DECIMAL(6);
A NAMED DATA ARC STATICALLY BINDS DATA PROCESSORS AND DATASETS. A DATA PROCESSOR HAS READ OR WRITE ACCESS TO A DATA SET IF THE ARROW POINTS TO OR FROM THE DATA PROCESSOR RESPECTIVELY. EXAMPLE: PROCESSOR P1 IS INITIATED BY CONTROL NODE N1. P1 READS DATA FROM DATASETS D2 AND D3 AND WRITES THEIR SUM INTO DATASET D1.		@DATA_GRAPH; @PROCESSORS P1(N1); @DATASETS D1, D2, D3; @ARCS DA1, DA2, DA3; DA3 (D3:P1); DA2 (D2:P1); DA1 (P1:D1); @END; PLIP INTERPRETATION @DATASET D1 FIXED BIN(31); @DATASET D2 FIXED BIN(31); @DATASET D3 FIXED BIN(31); @PROCESSOR P1; @READ(D3); @READ(D2); D1 = D2+D3; @WRITE(D1); @END PROCESSOR;