The Use of a Module Interconnection Language in the SARA System Design Methodology*

Maria Heloisa Penedo &
Daniel M. Berry &&

Computer Science Department
University of California at Los Angeles

ABSTRACT

In software design it is highly desirable to be able to deal with the structures of both the algorithm and the record of execution. A design methodology called SARA is being developed at UCLA. SARA's models are able to deal with the record of execution structure. This paper proposes the addition of a Module Interconnection Language to enable SARA to deal with the algorithm structure. An example illustrates how this combination can assist in the software design process.

Keywords: software design, module interconnection, MIL, algorithm, record of execution, modeling, realization.

Introduction

SARA (System ARchitect's Apprentice) [1, 2, 3] is a computer-aided system which supports a structured multi-level requirement driven methodology for the design of reliable (possibly concurrent) software or hardware digital systems. SARA is under continual development at UCLA. In this paper we consider only software design.

It is well known, in software, that errors are introduced during the design and implementation processes as a result of:

- inadequate statement of the requirements of the system,
- improper decomposition of a system into subsystems,
- inadequate testing of the system during the design process,
- inadequate care in the production of the implementation from the final designs.

SARA attempts to address this problem of design errors by providing effective means for synthesizing and analyzing a system and, furthermore, by providing a smooth and continuous path from programming-in-the-large (the act of decomposing a large software system into subsystems of modules) [4] to programming-in-the-small (writing code in some algorithmic programming language) [4]. It provides a designer with interactive tools and checking procedures to enforce consistency between requirements, structure, function and behavior. In recent work [5, 6], methods are proposed for developing test environments during the system design process.

The objective of this paper is to propose the incorporation of a Module Interconnection Language (MIL) into SARA. The addition of this capability reduces the gap between the model and realization of a system by providing for the derivation of code skeletons. Furthermore it provides SARA with a means for defining the algorithm (given by the MIL) and record of execution (given by SARA's existing models) structures which can assist the design and implementation processes by providing:

a) a better means for expressing the decisions made, by using software design methodologies such as composite design [7] and information hiding [8, 9],
b) a better description of the interfaces between modules,
c) a better means for managing teams, since the interfaces are well defined,
d) a better means for expressing code abstractions,
e) a better means for documenting systems.

In short, this capability should enhance the usefulness of SARA as a software design tool.

The first section discusses the necessity of distinguishing between the algorithm and record of execution structures in the modeling and design processes. Next a MIL is proposed for expressing algorithm structure. Section 3 describes SARA's Structural Model (SL1) and SARA's Graph Model of Behavior (GMB); we show how these models express the structure and behavior of the record of execution. It is then suggested how a MIL might

----

* This work was supported in part by the U.S. Department of Energy, Contract No. EY-76-S-03-0034, PA 214.

& Supported in part by the Conselho Nacional de Desenvolvimento Cientifico e Tecnologico, CNPQ, Brasil.

&& Supported in part by the Lady Davis Foundation, Israel. For 1978-79, Visiting at Faculties of Mathematics, Hebrew University, Jerusalem, Israel and Weizmann Institute, Rehovot, Israel.

CH1479-5/79/0000-0294$00.75 © 1979 IEEE

Reprinted from PROCEEDINGS OF 4th SOFTWARE ENGINEERING, September, 1979

profitably be incorporated into SARA. An example of a design using the MIL-extended SARA is carried out, in section 6, illustrating how code skeletons can be generated. This example also shows that the use of a MIL offers a degree of freedom which can lead to a clear exercise of design choices which might otherwise go unnoted.

1. Algorithm vs Record of Execution

Our attempts to use SARA for software design have met some difficulties. We found that we were unable to unambiguously express all degrees of freedom in selecting a module structure to impose on the code that has the required behavior. We found in particular that we could not express conveniently and in a graphical manner, design decisions such as:

1.1) which processes are instantiations of which procedures;

1.2) which data structures are instantiations of which data types;

1.3) which procedures and data types are visible to each other.

That these aspects of a program's structure can vary without the program's behavior or function also varying is a prime thesis underlying Parnas' Information Hiding Principle [8, 9] and Myers' Composite Design and Analysis [7].

Lessons learned from the history of programming language definition provided the clues as to how these degrees may be incorporated into SARA. These lessons point to the necessity of distinguishing between the structure of an algorithm (i.e., code) on one hand and the structure of the record of execution (i.e., instantiations) on the other.

For programming languages, this distinction was clarified with the introduction of the contour model [10] of block structured program execution and other similar models. In the contour model, each state (snapshot) of a computation consists of a reentrant time-invariant algorithm containing the code for the program being executed and a time-varying record of execution, which contains the processes and data cells which are allocated, modified, accessed and deallocated during execution.

In this scheme, a procedure is a named piece of code representing an algorithm. It contains parameter and local variable declarations and a sequence of instructions. A process (or activation) is an instantiation of a procedure which results from a call of the procedure. For each such instantiation of a procedure, a collection of data structures is allocated in the record of execution, one structure for each parameter and local variable declared in the procedure. A process uses one of these structures whenever its procedure says to assign to or access a parameter or local variable.

In the same manner, a data type definition (cluster [11], forms [12], etc) is a named piece of the algorithm comprising a template for the allocation of an element of the type plus a procedure for each operation of the type. A data object is an instantiation of a data type which results from the declaration or the allocation of a variable of that type. For each such instantiation of a data type, a data structure is allocated in the record of execution (according to the template). The instantiations of the operation procedures are created as these operations are called.

Separation of a process from its procedure is necessary to permit recursion and parallel processing. If there were no separation, for each procedure there would be but one process and thus but one copy of the cells for the variables declared in the procedure. Each recursive call (instantiation) and each parallel process (instantiation) of the same procedure would clobber each other as they all attempted to use the same cells at the same time. Similarly, separation of a data object from its data type definition is necessary to permit multiple structures of the same type. Were there no separation, assignment to one data structure would clobber all other structures of the same type. Thus, in general the algorithm must be separated from and distinguished from the record of execution.

We want to have the freedom to

a) decide on the structure of the procedures and data types in the algorithm, thus deciding item 1.3 above, and
b) decide which components of the record of execution are instantiations of which components of the algorithm, thus deciding 1.1 and 1.2 above.

This freedom of decision together with methods for expressing these decisions can be used to bridge the gap between modeling and code.

Thus it seems clear that any software design system and methodology must be able to deal with:

- algorithm structure,
- record of execution structure, and
- the correspondence between the elements of these structures.

Other authors have noted the distinction between the algorithm and the record of execution. But it seems that explicit use of this distinction and the degree of freedom it offers have not been incorporated into software design methodologies. Table 1 shows, for each of several software design methodologies, the correspondence of its terminology to our notions of procedure and data type definition in the algorithm, and process and data object in the record of execution.

Table 1. Methodologies and the correspondence between their terminology and our notions of algorithm and record of execution.

Methodology or Language	Algorithm		Record of Execution	
	Procedure	Data Type Definition	Process	Data Object
SADT[13]	?	?	Activities	Things
HIPO[14]	Hierarchy	?	Process	Input-Output
CDA[7]	Structure Diagram Functional Module	Structure Diagram Informational Module	?	Problem Structure
PSL/PSA[15]	?	?	System Structure	Input-Output and Data Structure
Jackson[16]	?	?	Program Structure	Data Structure
MIL[4]	Module	?	?	?
MIL[17]	Module	Module	?	?

SADT(TM) = Structured Analysis and Design Technique Methodology

HIPO = Hierarchy plus Input-Process-Output

CDA = Composite Design and Analysis

PSL/PSA = Problem Statement Language/Problem Statement Analyzer

MIL = Module Interconnection Language

2. What is a MIL

A Module Interconnection Language (MIL) - as proposed in the literature [17] - is a language for describing software definition module structure. Its main function is to establish the accessibility of resource names, i.e., identifiers, procedure names, type names, etc., among modules, and to assist in binding resource names to the modules which provide and need those resources. Other systems and languages such as DREAM [18, 19] and GYPSY [20] include constructs for defining module interconnection.

The MIL we are proposing consists of: MIL-modules, MIL-sockets and MIL-interconnections. A MIL-module represents a procedure or data type definition. Its name is made externally visible by means of a socket. A MIL-socket provides the interface between a module and the surrounding environment. Each MIL-socket names a procedure, a type, or an operation of a type. It tells whether the resource is offered or required. The attributes of a socket carry information about the named resource such as its parameter types, its return value type, if any, its specifications, etc. A MIL-interconnection is a directed arc connecting sockets, representing the accessibility of the name and the direction of the access.

For example, suppose a program PROG calls a subroutine SUB. The MIL model (graphical representation of a MIL definition) will be represented as in Figure 1.1, where PROG and SUB are distinct modules. Module SUB offers the resource SUB, as indicated by the socket name and the direction of the interconnection; module PROG requires resource SUB. The parameter types and the return value are specified in the socket attributes. The names of the connected sockets do not have to be the same.

Suppose also that a procedure PROG makes use of the read operation of a data type FILE. The MIL representation is shown in Figure 1.2, where PROG requires operation READ from data type FILE.

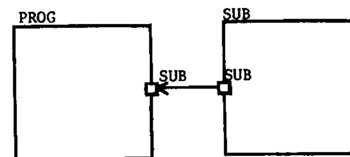


Figure 1.1. Example of a MIL model.

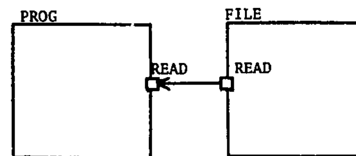


Figure 1.2. Example of a MIL model.

How useful is a MIL in the design process? A MIL can enhance both top-down-refinement and bottom-up-abstraction design methods. Top-down refinement means partitioning a system into subsystems. A MIL definition of the interfaces facilitates the task of splitting a big system into smaller ones, with the intent that they be modeled or programmed by different people, since it is known for each module what other abstractions are available for use and what must be offered for use. Bottom-up composition consists in interconnecting a set of predefined building block (off-the-shelf) models to form a model of a system being designed. This task is also more effective when, through the MIL capability, the modules to be connected have defined interfaces. It becomes possible to check, for each module, whether all resources it uses are available and compatible.

Furthermore, code skeletons come from MIL since their modules represent procedure or data type code structure, i.e., one code module per MIL-module.

Thus, a MIL deals with code structure and identifier visibility. It is this information that we want to see specified in the SARA methodology. In the next sections we describe some of SARA's models and their relationship with the MIL we are proposing.

3. SARA's MODELS

The SARA methodology supports both a top-down partitioning procedure (refinement) and a bottom-up composition procedure (abstraction) as illustrated in Figure 2. The SARA computer-aided system comprises a number of language processors and tools for assisting the designer using the SARA methodology. The language processors include interpreters which accept system descriptions in various languages and perform checks, displays, etc. Among these languages we distinguish: GMB, which itself contains languages for modeling behavior in three domains: flow of control, flow of data and interpretation, and SL1, which serves to describe hierarchically related structures to which the behavioral models can be mapped. These languages are discussed tersely below and further detail can be found in [21, 2]. There is a simulator [22, 23] providing an interactive simulation environment which permits experiments on the behavior models.

3.1. SL1 Structural Model

A structural model is mainly used to enforce modularity [24] by providing a better means to enforce encapsulation [9] and by permitting the isolation of parts of the system which then can be modeled and analyzed separately.

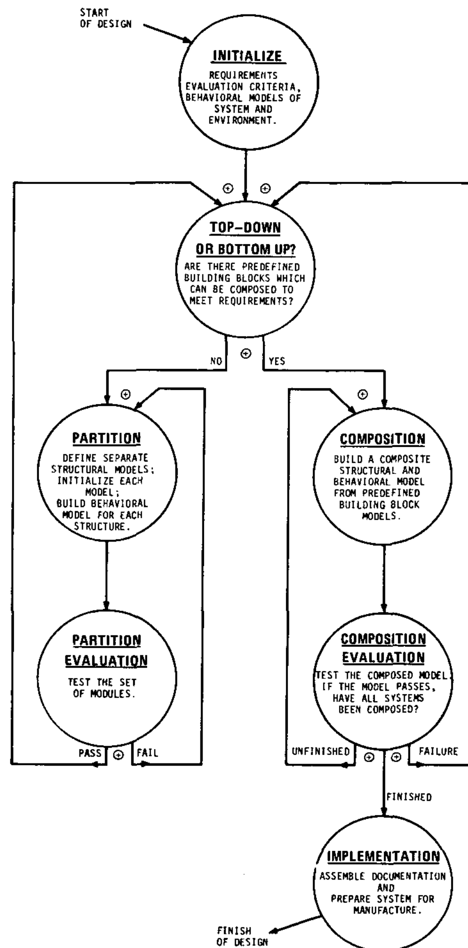


Figure 2. UCLA's SARA Design Methodology.

SL1 [25] is SARA's modeling language, designed for describing the structure of hierarchical modular systems. Used by itself, however, SARA's structural model has no behavior associated with it. It is used simply to define interconnected modules at various levels of abstraction and to allow the designer to specify a nested space of names to be used with the behavioral models. There are three kinds of structural elements: modules, sockets and interconnections. A module is used to encapsulate part of a behavioral model and abstract detailed behavior into sockets. A socket represents the interface between a module and its environment; it is always attached to a module or an interconnection. An interconnection connects modules at their sockets; it represents a potential flow of data or control which is made explicit only in a behavioral model. Furthermore, each element of the structural model is responsible for carrying out at least one of the requirements which drive the design of a system.

Figure 3a shows a structural model consisting of a module UNIVERSE which contains two sub-modules: ENVIRONMENT and SYSTEM. ENVIRONMENT has 3 sub-modules: SOURCEFILE, INITIATOR and OBJECTFILE. LX, LEX and GN are sockets representing the interface of the module ENVIRONMENT. Interconnections L1, L2 and L3 connect the modules ENVIRONMENT and SYSTEM.

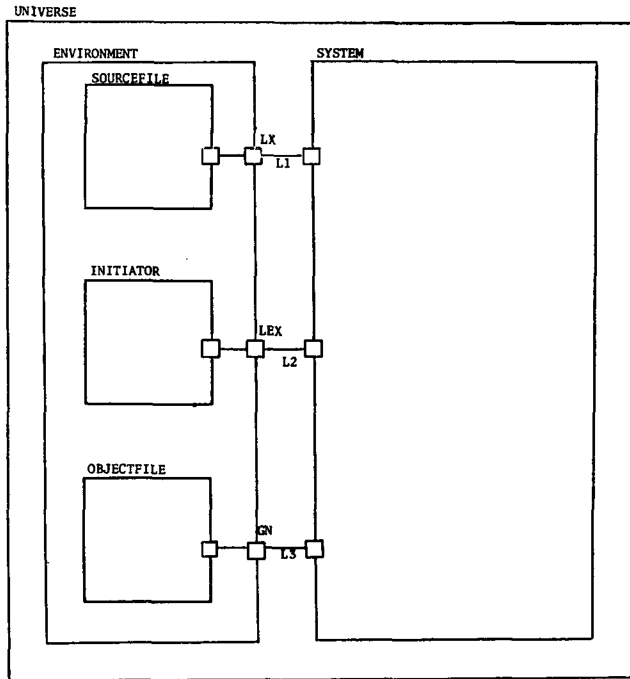


Figure 3a. Example of an SL1 model.

A module's internal structure may consist of modules, sockets and intèrconnections. An interconnection can also be refined into modules, sockets and interconnections. A socket can only be refined into sockets. Further detail can be found in [25].

3.2. GMB Behavioral Model

The behavioral model GMB (Graph Model of Behavior) [26] consists of two graphs: a flow-of-control (CG) and a flow-of-data (DG) graphs, together with interpretations associated with the nodes of the data graph.

Flow of control behavior can be expressed by control nodes, which span initiation and termination of associated processes, and by directed control arcs. Associated with each control node there is an input logic and an output logic. These logics, which are boolean expressions involving the arcs, express precedence and consequence conditions. Data flow is modeled in the data graph through processors and datasets, where the processors are responsible for the transformation of the data stored in datasets. There is a many-to-one function, mapping nodes in the control graph to processors in the data graph, with the property that each controlled processor (in the DG) must have at least one control node (in the CG) associated with it; for this reason processors in the data graph are called 'controlled processors'.

Figure 3b shows a GMB model: nodes are represented by circles; processors by hexagons; and datasets by rectangles. The control graph is expressed by nodes N1, N2 and N3 and the arcs S, A1, A2 and X connecting them. The output logic for node N1 is A1*A3 (A1 and A3); the input logic for N3 is A2*A3; the input logic for N2 is A2, etc. The data graph is expressed by processor P2 (associated to node N2) and datasets SOURCEFILE, OBJECTFILE and RETURN; the data arcs show the direction of the flow of data. The mapping between control nodes and data processors is shown in the picture by means of dotted lines.

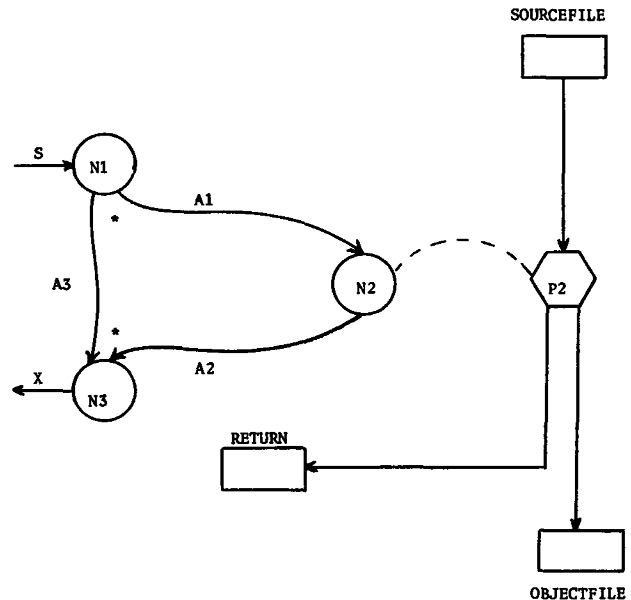


Figure 3b. Example of a GMB model.

GMB primitives may also be refined according to a set of defined rules [21, 28].

The meaning of a GMB model or graph is given through an abstract machine called the token machine for the graph. A state of the token machine is a placement of tokens on some arcs of the control graph together with the list of active nodes, and from any state the transition of the machine yields one next state. Thus, from a given initial state, the token machine appears to move tokens through the graph according to the transition rule described below.

A node is active if in some previous state transition it was initiated but in no subsequent transition it was terminated. A node may be

initiated in any state in which it is inactive and in which its input logic is satisfied. The input logic of a node is satisfied if, by assigning the value 'true' to each arc containing a token and the value 'false' to all other arcs, the input logic expression evaluates to 'true'. At any state either a node which may be initiated or a node which is active is selected nondeterministically to be initiated or to be terminated respectively. When a node is selected to be initiated, it absorbs tokens on its input arcs according to its input logic and becomes active. When a node is selected to be terminated, it is terminated and tokens are distributed on its output arcs according to its output logic.

An initial state of the graph in Figure 3b has a token on arc S. In this initial state, no node is active and only node N1 may be initiated. It is initiated and the token is absorbed leaving no tokens on any arc in the graph. When N1 terminates, according to its output logic A1*A3, two tokens are distributed: one to arc A1 and one to arc A3. At this point, only node N2 can be initiated since the input expression for node N3 is not 'true'. Upon termination of node N2, there is one token on arc A3 and one token on arc A2; thus node N3 can be initiated since its input logic evaluates to 'true'.

During any state transition of the token machine, if the control node N of the control graph is initiated, the data graph processor P associated with this node (if any) is considered to perform its algorithm. In Figure 3b, when node N2 is activated, processor P2 is considered to perform its algorithm, i.e., takes input from dataset SOURCEFILE, performs some transformation and outputs data to datasets RETURN and OBJECTFILE. We chose not to have any data processors associated with nodes N1 and N3.

PLIP (an extension to PL1) [27] is the language used for writing interpretations to be associated with the data graph; PLIP procedures provide the algorithms for the processors while declarations provide the type for the datasets. Whenever a data graph processor is considered to perform its algorithm, the token machine calls the PLIP procedure (if any) for this processor.

It is worth noting that the topology of both graphs in a GMB is fixed and that they are used to model pseudo-static aspects of system behavior. Any dynamic allocation of processes or datasets that cannot be expressed through the GMB fixed topology must be hidden inside a single node, e.g., the PLIP interpretation for this node is a recursive procedure or it causes dynamic allocation of data cells. In the process of designing any system comprising dynamic allocation, expansion of nodes into subgraphs will ultimately lead to a graph containing nodes hiding dynamic allocation which are not themselves expandable into subgraphs. This is an intentional restriction of the GMB model; at present all dynamic allocation is modeled in the interpretations attached to the unexpandable nodes. Another possible approach is the use of a GMB with a dynamically changing topology, i.e., a dynamic GMB. This is a topic for further research.

3.3. SL1-GMB relationship

As mentioned before, the SL1 structural model is mainly a space of names to be used with a behavioral model. An important aspect of SARA is the mapping between the behavioral and structural models:

- GMB submodels (control and data), uninterpreted or interpreted, are mapped to SL1 modules;
- GMB arcs crossing module boundaries are mapped to sockets (in the module) and the corresponding interconnections.

This mapping provides the SARA tools with means to detect cases of inconsistency and incompleteness in the design.

In Figure 3c, Figure 3b was mapped into Figure 3a. Node N1 (in the INITIATOR) starts the SYSTEM, i.e., it sends a token to node N2 which activates processor P2. P2 reads the data from dataset SOURCEFILE (mapped to module SOURCEFILE), performs some transformation on it, outputs the data to OBJECTFILE (mapped to module OBJECTFILE), while returning a status flag into dataset RETURN (in the INITIATOR). When P2 terminates, N2 sends a token to N3 (mapped to module INITIATOR) which stops the computation.

Which GMB subgraph is mapped to which SL1 module is very much a design decision (not fixed in

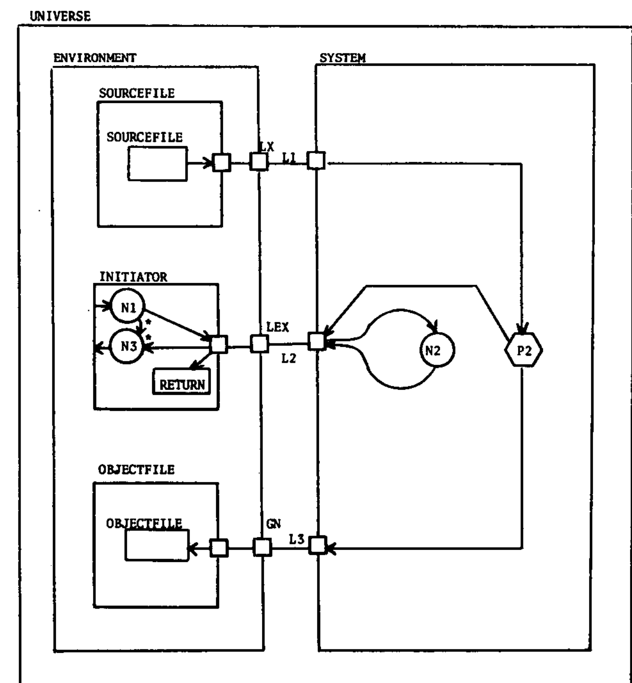


Figure 3c. Example of an SL1-GMB model.

the methodology); the choice of a particular subgraph to be mapped to a particular module is tantamount to abstracting the behavior of the subgraph into a single subsystem. Of course, however the decision is made, the behavior of the subgraph must fulfill the requirements which have been imposed on the module.

From now on we will use 'SL1-GMB model' to refer to the triple consisting of an SL1 model, a GMB model and a mapping between them.

4. SL1 x MIL

We now pose the question: Does an SL1 model represent a MIL model of a system?

In the terminology of section 1, each control node (or set of nodes) and associated data processor(s) models an instantiation of some procedure, and each dataset models an instantiation of some variable declaration. This follows from the very meaning of a GMB model. Values are considered to be stored in datasets, to be written or read from data arcs and to be processed by processors. It may, of course, be that two processors have the same interpretation, i.e., have the same PLIP procedure. It might also be that two cells have the same type, i.e., have the same attributes. In these cases the processors and datasets are instantiations of the same procedure and data type respectively.

Thus, given the strong mapping between GMB and SL1 and given that the GMB nodes represent instantiations, this implies that an SL1-GMB model represents a _structure of instantiations_. In contour model terms, an SL1-GMB model represents the structure of a completely pre-allocated record of execution.

As explained in section 2, a MIL represents the structure of the algorithm. Thus, an SL1-GMB model and a MIL model do _not_ convey the same information about software systems. In section 6, the compiler example shows an SL1-GMB model which can easily have more than one MIL model.

To illustrate the distinction between the SL1-GMB and MIL models, let us consider the case where a program PROG concurrently calls a subroutine SUB twice. Figure 4.1 shows the MIL model, which specifies that procedure PROG requires procedure SUB. Figure 4.2 shows the SL1-GMB model where instantiation PROG' calls instantiatiations SUB' and SUB'' in parallel.* The parallelism is represented by nodes N1, NS1 and NS2. The output logic of N1, A1*A2, indicates that upon termination of N1, one token is placed on arc A1 and one token on arc A2. A token on arc A1 permits node NS1

Figure 4.1. A MIL model corresponding to the SL1-GMB model of figure 4.2.

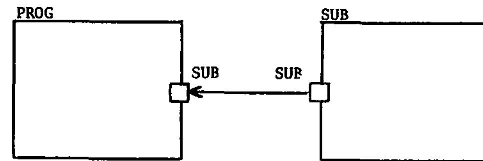
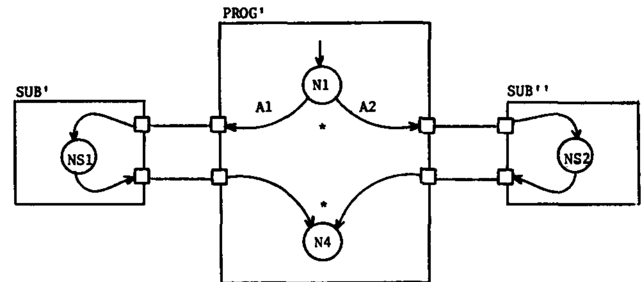


Figure 4.2. An SL1-GMB model of parallel instantiations of the same procedure.



(which models the procedure instantiation SUB') to be initiated; a token on arc A2 permits node NS2 (which models the procedure instantiation SUB'') to be initiated. This means that NS1 and NS2 can be activated at the same time, i.e.,in parallel. The SL1 modules SUB' and SUB'' represent instantiations of the MIL module SUB, which may run in parallel. Note that we chose not to show the data graph for this example.

The relationship between SL1 modules and MIL modules can be one-to-one, many-to-one or even zero-to-one. It is important to note that, when the system being modeled happens to have instantiations one-to-one with code [2] then one could get a code skeleton from the SL1-GMB models but this process does not generalize. Thus, code skeletons must come from MIL.

5. The Incorporation of a MIL

The single most important advantage of incorporating the information conveyed by a MIL into the SARA methodology is the help that the MIL and SARA can give to each other. While SARA provides a means for modeling and analyzing a system, the MIL will enhance its power by permitting the expression of decisions involving the structure of the algorithm and furthermore by providing a smoother path from modeling to code.

The combination of the SL1-GMB model, the MIL model, and the mappings between them permit the generation of even better code skeletons than is possible with either alone or without the mappings. Briefly, the SL1-GMB model identifies the variables (which name data type instantiations) and the calls

----

* We follow the convention of naming each instantiation of a procedure by its name followed by some unique number of apostrophes.

(which name procedure instantiations) of the code; the MIL model identifies the type and procedure definitions; and the mapping says which variable is of what type and which call is of what procedure.

Let us call the ability to specify the information conveyed by a MIL, a _Module Interconnection Specification Capability_ (MISC). The question remains as to how this capability is to be incorporated into the SARA methodology.

One approach* is the use of a MIL itself. This approach consists in defining a MIL model, in the way presented in section 2, and a mapping between the SL1-GMB model and the MIL model. That is, for an SL1-GMB model, the designer defines a MIL model and specifies the mapping between them. Each SL1 module will be mapped to at most one MIL module which is the definition of the particular instantiation. At a given level of the design, each SL1 socket and interconnection must be mapped to at most one MIL socket and interconnection which carries the visibility of the identifier whose invocation causes the control or data flow embodied in the SL1 interconnection.

In the next section, we explore the incorporation of the MIL approach into the methodology. We go through the top level of a requirement driven design of a compiler. This entails stating the requirements and then developing an SL1-GMB model for the compiler. At the appropriate point, the design decisions which are expressed by the MISC are made; they are expressed in the MIL approach. Then, from the SL1-GMB model, the chosen MIL model and the mappings between the two models, a code skeleton is derived for the compiler.

6. Example

To illustrate our discussion in the previous sections we use an example, a somewhat simplified design of a compiler. By using SARA's methodology, we go through the first steps (partition) of the system. Our objective is to show how we can express decisions involving the structures of the algorithm and of the record of execution and how code skeletons can be derived, i.e., passing from modeling to realization. For conciseness, several steps in the design process are omitted.

As mentioned before SL1 modules have no behavior. The GMB is a very powerful tool for analysis but as of now rules for mapping subgraphs into SL1 modules are not fixed; this is very much a design decision. Following the latest software design techniques, where modules are procedures or data type definitions, we are using pragmatics for modeling software in SL1-GMB. Then SL1-GMB modules are used to represent procedure instantiations or data type instantiations at whatever needed level of abstraction.

The assumptions and requirements for our system are as follows:

Environment Assumptions:

AE1. The environment must provide a file containing an input PASCAL program.

AE2. It must provide a file to contain the object program.

AE3. It initializes the compiler-system. Upon completion, the success or failure of the translation resides in the environment.

System Requirements

RS1. When requested to do so, the system reads an input program from the environment and attempts to translate it into a semantically equivalent object program.

RS2. If translation succeeds, it notifies the environment by returning the boolean value 'true' and it outputs the (360 machine language) object program.

RS3. If translation fails, it notifies the environment by returning the boolean value 'false'.

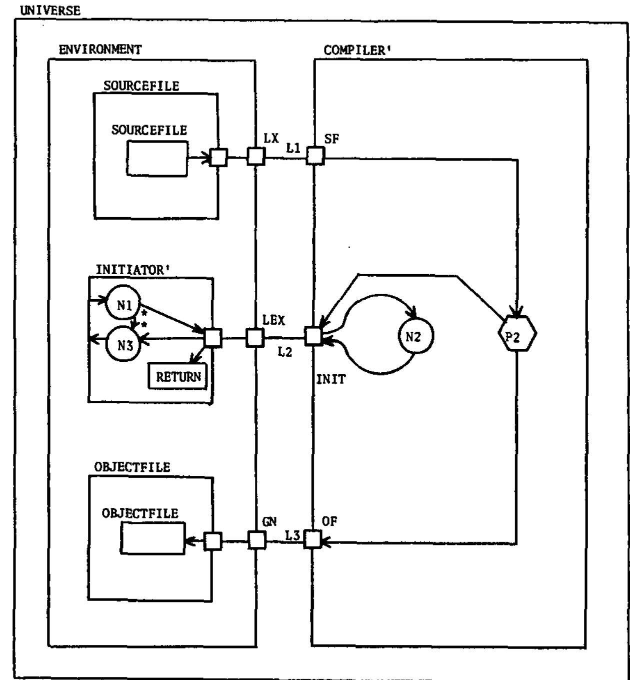


Figure 5.1. High level SL1-GMB model of COMPILER and its ENVIRONMENT.

---

* At least one other approach is known, the SL1-GMB-SOCKET Model Approach, but discussion of this approach is outside the scope of this paper; see [29] for more details.

The first step is to set up the environment we will be working with. According to the assumptions, there are three modules in the environment: SOURCE FILE, INITIATOR', and OBJECTFILE and we are designing the COMPILER' (Figure 5.1 shows the SL1-GMB model for it, which is the same picture as Figure 3c). Node N1 in the INITIATOR' starts the COMPILER', i.e., it sends a token to N2 which activates processor P2. P2 reads the program from dataset SOURCEFILE, compiles it, puts the output in dataset OBJECTFILE, and returns a boolean value to the INITIATOR' stating the result of the compilation. When P2 terminates, N2 sends a token to N3 which stops the computation.

In the next step, we decide to go top-down and refine the COMPILER' system into a set of modules as shown in Figures 5.2a and 5.2b. We show the SL1 partition, the GMB partition and the mapping in one step. The SL1-module COMPILER' was refined into sub-modules LEXER', SYNER', GENER', TOKENSEQ, TOKENTABLE, IDTABLE and POLISH. Observe that node N2 is refined into the GMB control graph composed of N21, N22, N23, N24, and N25 (Figure 5.2a). Processor P2 is refined into the GMB data graph composed of the controlled processors P21, P22, P23, P24, P25, the datasets TOKENSEQ, TOKENTABLE, IDTABLE, POLISH and the datasets RETURN for the returned value (Figure 5.2b). The association of the nodes in the control graph and the processors in the data graph is shown by the use of identical numbers, i.e., P21 is controlled by N21, etc. Also observe that socket INIT is refined into subsockets IN, OUT and RT. This is a characteristic of the multi-level properties of SL1 and GMB models.

The INITIATOR' sends a token to the LEXER' activating node N21 which triggers the activation of processor P21, responsible for the lexical analysis; P21 updates datasets TOKENSEQ and TOKENTABLE using the information in SOURCEFILE. If lexical analysis fails, node N21 sends a token to N25, activating processor P25 which terminates the compilation and returns the boolean value 'false' to the INITIATOR'. Otherwise the LEXER' sends a token to the SYNER' activating node N22 which triggers processor P22, responsible for the syntactical analysis; P22 updates the IDTABLE and

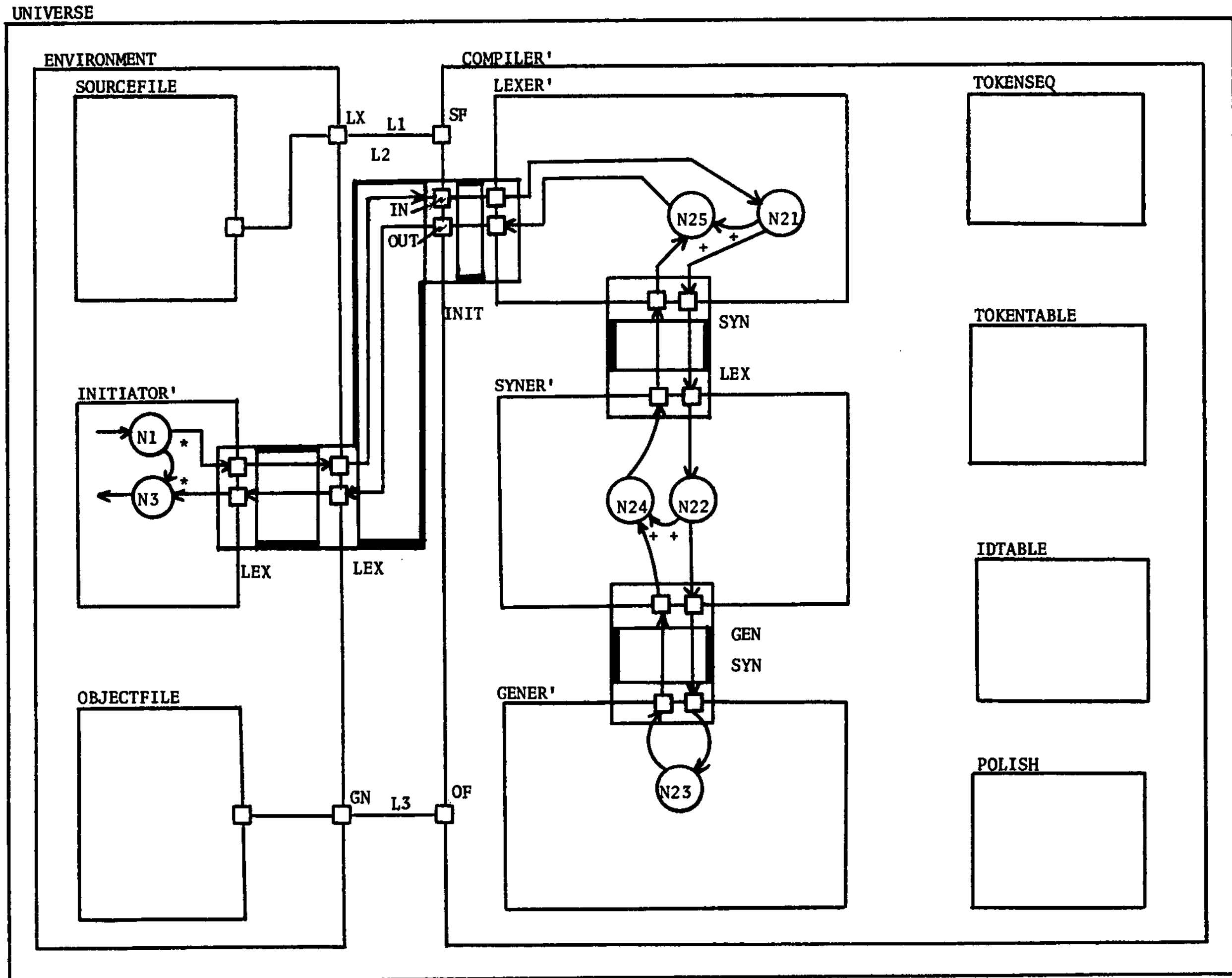


Figure 5.2a. SL1 partition and control graph refinement of Figure 5.1.

POLISH datasets using the information in TOKENSEQ and TOKENTABLE. If syntactic analysis fails, N22 sends a token to N24, P24 returns 'false', and control flows back to the INITIATOR' to terminate the compilation with the value 'false' returned. Otherwise SYNER' sends a token to node N23 in GENER'; this activation triggers processor P23; P23 generates the object code using the information stored in IDTABLE, TOKENTABLE, and POLISH and outputs it to OBJECTFILE. Regardless of whether or not code generation succeeded, when P23 is finished, control flows back through N24 and N25, and a boolean value indicating success or failure is returned, through P24 and P25, to the INITIATOR' at which time the whole computation is ended. Note that we have gathered together as subsockets the sockets representing a procedure call or a procedure being called.

All processors may or may not, at this point, have interpretations (written in PLIP), which explicate the data graph processor's functions, associated with them. For brevity, we choose not to write them. An example of multilevel design of software including interpretation can be found in [2].

Continuing the design process, we may want to refine once more one or more SL1-GMB modules. However it is reasonable at this point to try generating a code skeleton from the information present in the SL1-GMB models. We show how this can be accomplished and in the next step we show how the MIL model, mapped to this SL1-GMB model, provides for a more complete structure of the code.

A code skeleton generated from the SL1-GMB model can be as follows. We chose to represent all data as external variables or returned values of functions.

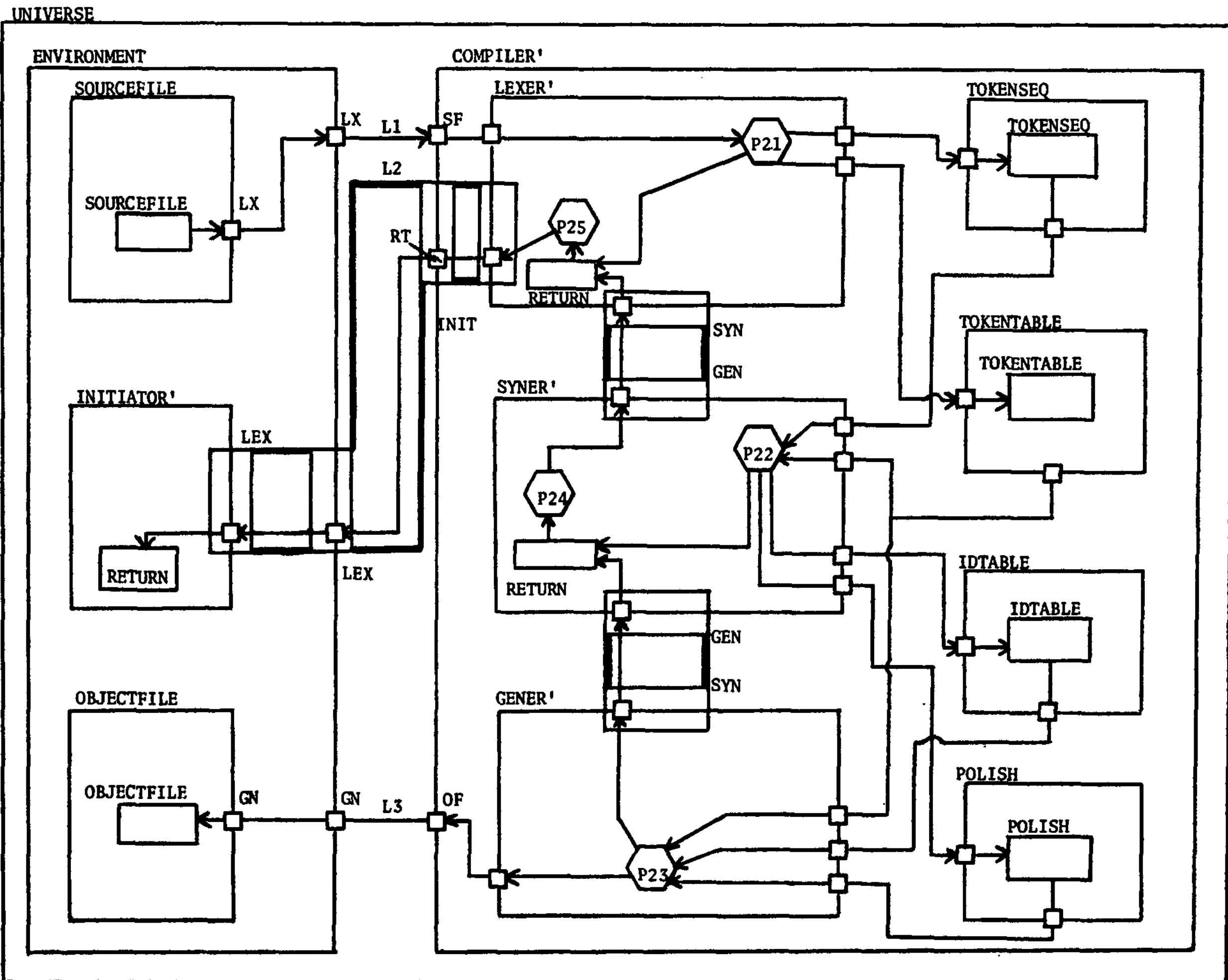


Figure 5.2b. SL1 partition and data graph refinement of Figure 5.1.

```
global  sourcefile: <sourcefile type>,
        objectfile: <objectfile type>,
        tokenseq:   <tokenseq type>,
        tokentable: <tokentable type>,
        idtable:    <idtable type>,
        polish:     <polish type>;

compiler:lexer: procedure() bool;
    external  sourcefile: <sourcefile type>,
              tokenseq: <tokenseq type>,
              tokentable: <tokentable type>,
              syner:  procedure;
    ...........
    call syner;
end lexer;

syner:  procedure()bool;
    external  tokenseq: <tokenseq type>,
              tokentable: <tokentable type>,
              idtable:  <idtable type>,
              polish:  <polish type>,
              gener:  procedure;
    ...........
    call gener;
end syner;

gener:  procedure()bool;
    external  tokentable:  <tokentable type>,
              idtable:  <idtable type>,
              polish:  <polish type>,
              objectfile:  <objectfile type>;
    ..........
end gener;
```

Using our pragmatics, explained in the beginning of this section we opted for considering all internal SL1 modules as external procedures or external data (global). Thus LEXER', SYNER' and GENER' are instantiations of procedures. In the code skeleton, each procedure specification includes an empty parameter list, a return value type, and the list of non-local (external) variables and procedures that it uses.

SOURCEFILE, TOKENSEQ, ..., etc are variables. In a previous publication [2], the types of variables and parameters were extracted from the dataset definitions in PLIP. Here we take a different approach since we want to consider these attributes as abstract types. Thus, we assume that each distinct variable has its own type and for the variable or parameter x we say that its type is <x type> since the type is not known at this point.

Continuing the design process, one may also want to identify which processes are instantiations of what procedures and which datasets are instantiations of what data types. This is exactly the MIL information.

We determine that LEXER', SYNER' and GENER' are instantiations of the procedures LEXER, SYNER, and GENER respectively. In the interest of saving work and decreasing the number of code modules to write, we examine the variables trying to identify single abstract types for several variables. TOKENTABLE and IDTABLE both have the properties of a table, and we decide that they both are instantiations of a single type constructor TABLE with sufficient parameterization and enough different operations to provide both required behaviors. Examination of TOKENSEQ and POLISH show that both have properties of a sequence. Thus we make them instantiations of a single type constructor SEQUENCE. In doing this we have identified also the necessity of a type TOKEN to serve as a type parameter to the type constructors introduced above.

Expressing these decisions in the MIL approach, explained in section 2, we come up with a MIL model as given in Figure 5.3. All the resources available and required are specified by the socket names and the direction of the interconnections. INPUTFILE is the type for the variable SOURCEFILE; OUTPUTFILE is the type for the variable OBJECTFILE. TOKENTABLE and IDTABLE are both instantiations of the type TABLE; TOKENSEQ and POLISH are both instantiations of type SEQUENCE. The module TABLE in the MIL representation represents the code (type definition and operations) for type TABLE. As such, it is required by every module which will be making entries into or looking up entries in either table. Similarly, the module SEQUENCE represents the code for the type SEQUENCE and is required by every module which inserts into or removes from either sequence.

The mapping for this example between the SL1-GMB models and the MIL model is:

a) one-to-one when referring to the procedures;
b) IDTABLE and TOKENTABLE in the SL1-GMB model are mapped to TABLE in the MIL model;
c) POLISH and TOKENSEQ in the SL1-GMB model are mapped to SEQUENCE in the MIL model;
d) SOURCEFILE and OBJECTFILE in the SL1-GMB model are mapped respectively to INPUTFILE and OUTPUTFILE in the MIL model.

Note that there is no SL1 module mapped to TOKEN in the MIL model.

From the additional information provided in the MIL model, the previously given code skeleton can be filled in and optimized. The result follows below. Now all variables have a type as specified by the mapping above; that is, variable TOKENTABLE is of type TABLE, etc. All types are also defined; this may be a good time to define the operations for each type.

```
global  sourcefile: inputfile,
        objectfile: outputfile,
        tokenseq: sequence(token),
        tokentable: table(token, string),
        idtable: table(token,
                    struct(n1:int,disp:int,
                           typename:string)),
        polish: sequence(token);
```

```
compiler:lexer:  procedure()bool;
    external inputfile, sequence,
             table, token : type,
             syner: procedure,
             sourcefile: inputfile,
             tokenseq: sequence(token),
             tokentable: table(token,string);
    ...........
end lexer;

syner:  procedure()bool;
    external table, token, sequence: type,
             gener: procedure,
             tokenseq: sequence(token),
             tokentable: table(token,string),
             idtable: table(token,
                          struct(n1:int, disp:int,
                                 typename:string)),
             polish: sequence(token);
   ............
end syner;
```

```
gener: procedure()bool;
    external table, token, sequence,
             outputfile: type,
             tokentable: table(token, string),
             idtable: table(token,
                          struct(n1:int, disp:int,
                                 typename:string)),
             polish: sequence(token),
             objectfile: outputfile;
    ..............
end gener;

inputfile:  type
            .......
end inputfile;
outputfile:  type
             .......
end outputfile;
table:  type(domain:type, range:type)
        .......
end table;
sequence:  type(element:type)
           .......
end sequence;
token:  type
        .......
end token;
```

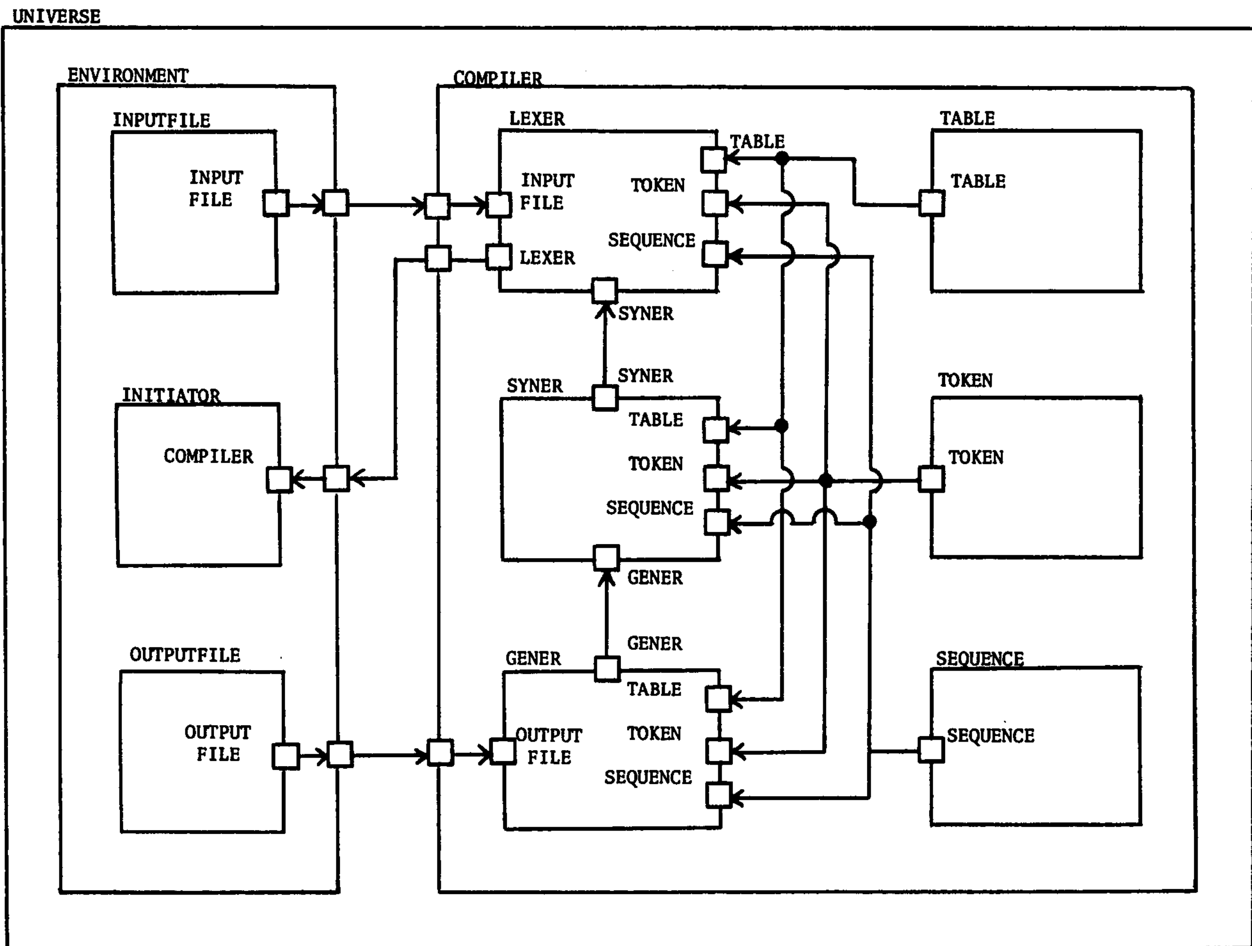


Figure 5.3. A MIL model of COMPILER.

By means of the example we showed how SARA may be used in the design of software to point to and to permit expression of decisions involving the structures of the algorithm and of the record of execution with the ultimate goal of providing a smoother path from modeling to code.

As mentioned before and illustrated by the example, the SL1-GMB model identifies the variables (which name data type instantiations) and the calls (which name procedure instantiations) of the code; the MIL model identifies the type and procedure definitions; and the mapping says which variable is of what type and which call is of what procedure.

It is worth noting that once the SL1-GMB together with the MIL model and the mapping between them are defined (by the designer), automatic generation of code skeletons may be performed.

7. Conclusions

It is widely accepted that design and implementation methodologies are a necessity. However there exists a gap between modeling and implementation, a gap which must be narrowed and, if possible, removed completely.

In this paper we attempted to attack this problem. We have discussed some of the reasons for incorporating a module interconnection specification capability (MISC) into the SARA methodology with the primary goal of providing a smooth and continuous path from programming-in-the-large to programming-in-the-small. We have demonstrated the importance of distinguishing between algorithm and record of execution structure in modeling and designing software. Modeling without this distinction is incomplete, and design without it leaves a number of design possibilities unconsidered and decided by default. We have shown how the existing SARA tools could benefit from the addition of a module interconnection language (MIL) to be able to express design decisions involving both the algorithm and the record of execution structures. Finally, we have indicated, with the aid of an example, how a MIL-extended SARA may be used in the design of software to point to and to permit expression of decisions involving the structures of the algorithm and the record of execution and how code skeletons can be generated from the models.

We do not claim, however, that a MISC, by itself, will solve all problems. Many open questions remain:

-- If an insufficiently structured model of behavior is used, we may not be able to find a path to code. One of the authors of this paper, is defining a Structured GMB, as part of her Ph.D. dissertation. It will include high level primitives such as procedure and cluster instantiations. Pragmatics for software design are also being developed such that a MISC, in conjunction with SARA's models will lead naturally to code.

-- Another question is how to define the MISC and provide associated tools so that it can be naturally integrated into the SARA system, or how to modify the SARA system to take full advantage of a demonstrably strong MISC. We have described one approach in this paper. However this approach may turn out to excessively increase the modeling burden on the designer. For this reason we are also exploring a way to represent the MISC information in a Socket Model, to be used in conjunction with the SL1-GMB model. This socket model would not only contain the MIL information but it might also include information such as statistics, timing, performance, etc.

-- There is also the problem of determining the consistency between an SL1-GMB model and a MIL model mapped to it.

-- We showed that a MISC can help in the generation of code skeletons; but the problem of translating the interpretations (written for the processors) into code, still exists. If the interpretation language (for modeling) is compatible with the realization language, as described in an earlier paper [2], this is an easier task; if not, the gap is still very large. We tend to think that the only solution to this problem is to have specialized interpretation languages for distinct implementation languages.

-- SARA's GMB model is very suitable for modeling concurrency. The example shown was a sequential one; other problems arise in a concurrent environment. What are the problems involved in mapping SL1-GMB models to MIL models and in translating the synchronization expressed in the graph into a programming language?

These and other questions are being analyzed as part of our continuing research into the subject of this paper.

BIBLIOGRAPHY

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

2 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).

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

4 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).

5 Drobman, J., "Building Block Methodology for Composition of Microprocessor-Based Digital Systems", Ph.D. Dissertation, Computer Science Department, UCLA (June 1979).

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

7 Myers, Glenford J., Composite/Structured Design, Van Nostrand Reinhold Company, NY (1978).

8 Parnas, D.L., "A Technique for Software Module Specification with Examples", CACM 15:5, pp. 330-336 (May, 1972).

9 Parnas, D.L., "On the Criteria to be Used for Decomposing Systems into Modules", CACM 15:12, pp. 1053-1058 (December, 1972).

10 Johnston, J. B., "The Contour Model of Block Structured Processes", Proceedings of ACM Conference on Data Structures in Programming Languages, SIGPLAN Notices, 6:2 (February, 1971).

11 Liskov, B., A. Snyder, R. Atkinson and C. Schaffert, "Abstraction Mechanisms in CLU", CACM 20:8, pp. 564-576 (August, 1977).

12 Wulf, W.A., R. L. London and M. Shaw, "An introduction to the Construction and Verification of Alphard Programs", IEEE TSE : SE-2:4 (December, 1975).

13 Dickover, M.E., C.L. McGowan, and D.T. Ross, "Software Design using SADT", Proceedings of the 1977 Annual Conference of ACM, Seattle, Washington, pp. 125-133 (October, 1977).

14 Katzan, H. Jr., System Design and Documentation: An Introduction to the HIPO Method, Van Nostrand Reinhold Co., NY (1976).

15 Teichroew, D., and E.A. Hershey, III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis on Information Processing Systems", IEEE Transactions on Software Engineering, Vol.3, No.1, pp. 41-48 (January 1977).

16 Jackson, M.A., Principles of Program Design, Academic Press, London (1975).

17 Thomas, J.W., "Module Interconnection in Programming Systems Supporting Abstraction", Ph.D. Dissertation, Division of Applied Mathematics, Brown University, Providence, R.I. (April, 1976).

18 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).

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

20 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).

21 Campos, Ivan M., "Multilevel Modeling for Synthesis of Reliable Concurrent Software Systems", Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles (1977).

22 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).

23 Razouk, R.R., "GMB Simulator System Reference Manual", Computer Science Department, UCLA (January, 1977).

24 Dennis, J.B., "Modularity", Advanced Course on Software Engineering, Springer-Verlag, Vol. 81, Chapter 3A, pp. 128-182.

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

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

27 Overman, William, "PLIP Reference Manual", Computer Science Department, UCLA (July, 1977).

28 Ruggiero, W., "A Distributed Data and Control Driven Machine - Programming and Architecture", Ph.D. Dissertation, Computer Science Department, University of California, Los Angeles (1978).

29 Penedo, M. H and D.M. Berry, "The Use of a Module Interconnection Specification Capability in the SARA System Design Methodology", Computer Science Dept, UCLA (July, 1978).