Applications of Program Understanding and Rule-Based Quality Assurance to Slam II Simulation Programs

by

N. Rodney Wendt

A thesis
presented to the University of Ottawa
in fulfillment of the
thesis requirement for the degree of
Master of Science
in

Computer Science
Ottawa-Carleton Institute for Computer Science
Department of Computer Science
University of Ottawa
Ottawa, Ontario, Canada.

ISBN 0-315-89691-4

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

I hereby declare that I am the sole author of this thesis. I authorize the University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Norman Rodney Wendt

I further authorize the University of Ottawa to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Norman Rodney Wendt

To mom and dad

Acknowledgments

The author would like to thank Dr. Tuncer Ören, Dr Lou Birta, Dr. Doug King, Yves Dorego, Dr Osman Abou-Rabia, and Dr Martin Hitz for their efforts and advice during the development of E/Slam. The author would also like to thank Dr. Tuncer Ören for his guidance and support during the creation of this thesis. E/Slam has been carried out as part of a broad project funded by Bell Canada. I wish to express my appreciation for this financial support.

Abstract

With the advance of time, our inventory of simulation programs has and continues to accumulate. To maximize the return on our investment of time and money into these software systems, it is advantageous for us to reuse software components as much as possible. For example, previously engineered simulation models can often be reused and exercised under a new set of experimental conditions. Before a software component can be reused, the analyst must learn and understand its functionality. This learning process is often made unnecessarily difficult due to incomplete documentation. Another contributing factor is the complexity brought about by interacting directly with the program code. Furthermore, when it comes time to make updates to the code, the potential arises for semantic and syntactic errors to work their way into the program. Knowledge-based program understanding systems with built in quality assurance can be used as an environment for simplifying the learning and the update processes, while ensuring an acceptable degree of quality has been maintained during the update process. This thesis discusses program understanding and quality assurance issues related to the Slam II programming language and discusses the architecture of E/Slam (Elucidation of Slam II programs). E/Slam is a knowledge-based program understanding system with built-in quality assurance ability.

List of Figures

Table of Contents

Detailed Table of Contents

1 Introduction

The goal of this thesis is to explicate how program understanding systems can be used to enhance understandability and reusability of simulation programs. The author's aim is to demonstrate how the concept of program understanding systems can be taken two steps further, first by providing an environment for simulation experimentation, and second by certifying the correctness of such changes through quality assurance mechanisms that are an integral part of the system environment.

To justify program understanding systems with built-in quality assurance, we will first discuss the steps comprising the simulation program lifecycle. Then we will take a look at some of the problems that are known to exist in the development lifecycle. The body of the thesis will demonstrate how a program understanding system with built-in quality assurance can address the cited concerns. These concepts will be demonstrated with reference to two systems called E/Slam (Elucidation of Slam II programs) and QA/Slam (Quality Assurance of Slam II programs) developed at the University of Ottawa.

E/Slam is a system designed to elucidate Slam II simulation programs. Consequently, background information about Slam II is offered in Chapter 2. This will be drawn upon in subsequent chapters when demonstrating how the program understanding and quality assurance process is carried out in E/Slam and QA/Slam. Chapter 3 and 4 describe the architecture of E/Slam. Chapter 5 discusses Slam II program understanding issues. The idea here is to give a detailed description of the various Slam II information sources that must be tapped before a Slam II program understanding system can elucidate program

information. Chapter 6 introduces and describes in detail, the template approach for elucidating Slam II program information. Here, a detailed description is given of documentation template concepts and E/Slam's statement templates and program templates. This chapter also describes how information is gleaned from the program and combined to produce documentation template entries. Chapter 7 expands on the documentation templates introduced in Chapter 5 and 6 by demonstrating how templates can be used not only as a structure for displaying information extracted from a program, but also as a structure for making updates to programs. Chapter 7 also demonstrates the need for such a system to be able to generate updated versions of program source code to reflect program changes made form within the program understanding system environment. Chapter 8 describes the quality assurance component of E/Slam called QA/Slam. QA/Slam is an embedded knowledge-based system for the quality assurance of Slam II programs. The thrust of Chapter 8 is to demonstrate how AI techniques can enhance software tools by analyzing programs for inconsistencies in specification, by offering advice on how to correct inconsistencies discovered, and by certifying the correctness of programs when no inconsistencies are discovered.

1.1 Simulation Program Lifecycle

The goal in computerized simulation is to model the dynamic behavior of some aspect of a system under study, for the purpose of performing goal directed experimentation on the model. During the course of a simulation study a conceptual model is transformed into a simulation model. The simulation model is driven under a set of experimental conditions to produce simulation results that

are used for decision support purposes. The reader is referred to (Ören, Zeigler 1979, p69–82) for a detailed discussion of the distinction between simulation model and experimental conditions. At the end of a simulation study the model and experimental conditions may undergo a redefinition process. Balci (1992) mentions that the redefinition process is entered when it is necessary to

- update the model to represent the current form of the system,
- alter the experimental conditions to obtain another set of results,
- perform maintenance,
- modify the model for other uses, or
- define a new system model.

Before any modifications can be made to the model, one must have an understanding of its functionality. This is necessary to know where and how the desired changes should be manifested in the program. Once these questions are answered, the analyst can proceed to make the necessary updates. Unfortunately, the task of learning and updating simulation programs has its problems, some of which are discussed below.

1.1.1 Insufficient Documentation

A substantial inventory of simulation software has accumulated over the years. At some point in the development lifecycle, it often becomes apparent that there is a serious lack of understanding about important features of the existing software. This is often due to unclear, incomplete or lost documentation. In these situations one has little recourse, because the original developers of the software are often

unavailable for consultation about the design, organization and even the proper usage of the software (Birta et al. 1991).

1.1.2 Need for Tools to Enhance Understanding and Updating of Simulation Programs

The situation often arises where one who is not an expert in a simulation program's implementation language, wishes to drive the model under a new set of experimental conditions. For such a user, gaining an understanding of the program can be an arduous task. Although specifying new experiments for an existing simulation model is far less difficult than developing the entire simulation model, the syntactic and semantic complexity of some simulation languages make it difficult for one to confidently specify new experiments, unless one has an intermediate level of expertise in the language.

Although it may be true that program changes required to specify new simulation experiments are small and localized, one must have an understanding of the simulation program from a global perspective to know where to make the necessary changes, and to be confident that the changes made do not conflict with specifications elsewhere in the program. Traditionally, this required one to have a reasonable understanding of the simulation language in which the program was developed. This language barrier between the analyst and the program code makes learning difficult and drastically reduces the size of the audience that can interact with the program model and run their own experiments.

1.1.3 Regression Testing

The correct procedure following any program change is to run a series of tests to determine if the change has caused other aspects of the program to regress. This process is called regression testing, and it is an important task because programming changes and error corrections tend to be much more error prone than the original coding of the program (Myers 1979). The reader is referred to (Whitner, Balci 1989) for a comprehensive list of simulation model testing techniques applicable to programmed model verification and validation.

In practice it appears these verification and validation steps are not taken as seriously as they should be (Nielson 1991). The reasons usually sited are that the changes are very small and thus do not require testing, or that testing is uneconomical. This kind of oversight has an obvious negative affect on reliability and on the assurance of quality during the update process.

1.2 Addressing The Issues

A category of software processing, called software understanding has emerged to deal with the issue of enhancing program understandability (Ören 1992). "Software understanding is a generic term for a family of related concepts that are concerned with providing an enhanced perception of an existing software product" (Ören et al. 1992). This enhanced understanding may be a goal in itself. Alternately, it may simply serve as an essential prerequisite for a maintenance activity that needs to be undertaken.

In order to avoid a deterioration in the usefulness of our existing software resources, it has become increasingly important to develop tools that assist in

elucidating relevant aspects of software. However, this is only part of the picture. Equally important is the need for tools that assist in making updates to programs and tools that assure a certain degree of quality is maintained through the update process. In simulation, tools for updating programs include those designed for analyst who specifies and runs simulation experiments, and tools that aid in updating the simulation model.

Program understanding systems address the issue of incomplete documentation, since they go directly to the program code and build an understanding from there. These systems could be used to generate written documentation from the program source code automatically (King et al. 1992), (Ören et al. 1991). They could also be designed as interactive on line systems. Furthermore, since program understanding systems act as an intermediary between the user and program source code, they shield the user from unnecessary implementation details.

Program understanding systems with update capabilities reduce the need for one to have an intermediate or advanced understanding of the implementation language, since these systems offer assistance in both the learning and update phase of the lifecycle. Because these systems handle information flow in two directions (i.e., from source code to user and from user to source code), theoretically the user is not required to interact directly with the code.

Program understanding systems having update capabilities combined with built-in quality assurance mechanisms address the concern about programs regressing following a software update. The quality assurance mechanisms ensure program changes do not cause the overall system to enter an inconsistent state. Quality is assured by performing analysis on the simulation program from a global

perspective, taking into account program changes specified by the user. The goal is to determine if the program change introduces error conditions, warning conditions or anomalies. If an update has caused the program to regress, the user is notified of the situation. This kind of mechanism could also be designed to give advice on how to alleviate inconsistencies in specification that have been discovered.

A program understanding system called E/Slam and a quality assurance system called QA/Slam have been developed at the University of Ottawa to demonstrate how the aforementioned concerns can be addressed. E/Slam is a step toward a simulation environment that has software understanding abilities combined with an interface through which simulation programs can be studied and simulation experimentation conditions can be updated. QA/Slam ensures Slam II programs updated using E/Slam maintain a respectable degree of quality. AI techniques are used to ensure program modifications introduced using E/Slam have not caused the overall program to enter an inconsistent state. We will study these two systems in more detail as we progress.

2 The Slam II Simulation Language

E/Slam and QA/Slam are systems for understanding Slam II programs. Consequently, it is useful that we begin with a background discussion of the Slam II language.

2.1 Slam II Modelling Formalism

Slam II (Simulation Language for Alternative Modelling), is a simulation language that provides three alternatives for modelling systems (Pritsker 1986). Systems can be specified from a network system perspective, continuous system perspective, discrete event perspective or any combination of the three.

Network System Perspective

The network system perspective is realized through a collection of network symbols or nodes, which are interconnected by activity arcs. The resulting graphic representation is called the network model, and can be realized as one or more disjoint directed graphs.

During a simulation run, objects called entities flow from one node to the next, through the network graph. Each network symbol in the language has a different functionality associated with it and this functionality is invoked when an entity arrives at the node. An entity can have a set of attributes associated with it, that hold information specific to that entity. Each node's function is parametric in nature and the language allows for parameter values to be supplied by global system variables, constants, function calls, or the attribute value of an entity that

arrives at the node. These values are bound to the node's parameters when an entity arrives at the node. Data is made globally accessible to the program by using Slam II system variables. The system variables are visible to all network statements, control statements, and Fortran subroutines comprising the simulation program. Network and control statements will be discussed shortly.

Through the use of the network nodes, the modeler has the ability to create side effects on entity attributes, system variables, storage files, gates, and resources. Storage files act as holding places for entities and data. Since only one entity can be serviced by a node at a time, some nodes have the ability to queue arrivals of entities. The storage file serves as a holding place for these queued arrivals.

Gates and resources are central to the process oriented simulation component of Slam II. Gates are used for regulating the flow of entities in other parts of the network model. A resource is a supply that can be drawn upon by the simulation model. All resources are defined in the same way. The meaning imposed upon them by the modeler distinguishes a resource as being one kind as opposed to another.

Network modelling can be carried out through the use of a graphical environment called TESS (The Extended Simulation System), which is an extension to Slam II (Standridge et al. 1987). TESS is used to build a program model through a set of network symbols. A network graph is created by interconnecting the various kinds of network symbols. The network model is then mapped onto a set of Slam II program statements, thus producing an equivalent Slam II statement model. The Slam II statement model will be discussed shortly.

Continuous System Perspective

The second modelling view supported by Slam II is continuous system modelling. To utilize this approach the modeler defines, in a Fortran subroutine, the difference equations and/or differential equations of the continuous system to be modelled. Through Slam II control statements the modeler specifies details such as the number of equations, step size, and data recording intervals for the system.

Discrete Event Perspective

The third modelling formalism supported by Slam II is discrete event modelling. This modelling approach is realized through the use of Fortran subroutines and the Slam II event calendar. The event calendar is responsible for scheduling events. When an event is at the top of the calendar its corresponding Fortran subroutine is executed. Slam II provides a set of pre-written Fortran subroutines for carrying out various discrete event operations. This includes subroutines for event scheduling, file manipulation, random sampling and data collection.

E/Slam and QA/Slam are designed to aid in understanding Slam II programs modelled using the network system perspective. Documentation of the Fortran component of Slam II is handled by two separate tools called OrFor (Ören, King 1989), and MIA (Ören et al. 1989). OrFor has been integrated with E/Slam.

2.2 Slam II Program Components

A typical Slam II program, has three major components: network statements, control statements and Fortran subroutines. Each is discussed in turn below.

Network Statements

The network statements are used to build a node graph model. There are three categories of network nodes: gate related, resource related and general network flow related. Gate related nodes, denoted by α in **Figure 2.1** are in some way related to gates either by controlling the gates status or by having its behavior influenced by gate status.

Similarly, resource related nodes, denoted by β in **Figure 2.1**, either influence, or are influenced by resources defined in the model. These nodes can either modify the status of a resource, or their operation can be directly affected by the availability of a resource.

General network flow related nodes have varied functionality and generally speaking can not be classified into a particular category, except that they all contribute to and influence the flow of network entities in the model. Nodes in this category are denoted by γ in **Figure 2.1**.

The Slam II language defines an ACTIVITY arc that is used to interconnect network nodes. Through the use of activity arcs the modeler is able to assign weights, conditions and probabilities to the arcs connecting network nodes. The weights are expressed as propagation delays and the conditions and probabilities are used for determining whether the associated arc will allow an entity to cross.

Control Statements

The control statements are used for specifying values relating to the pre-study, pre-run, post-run and post-study activities of the simulation study. There are three possible classifications for Slam II Control statements: initialization related,

ACCUMULATE	γ	Allows a specified number of entities to accumulate at the node and releases one entity when this specified number is reached.
ALTER	β	Alter the number of units of a resource available for allocation.
ASSIGN	γ	Assign a value to a variable.
AWAIT	α β	Wait for the availability of a specified resource, or wait for a specified gate to open.
BATCH	γ	Combine arrived entities into one composite entity.
CLOSE	α	Close the specified gate.
COLCT	γ	Collect the specified statistics.
CREATE	γ	Create entities for routing through the network.
DETECT	γ	Describe the conditions for a state event.
ENTER	γ	Entry point for entities to be injected into the network.
EVENT	γ	Interface for calling user written event code.
FREE	β	Free up units of the specified resource.
GOON	γ	Continuation node where every entity passes directly through.
MATCH	γ	Match entities residing in specified QUEUE nodes that have equal values of specified attributes.
OPEN	α	Open the specified gate.
PREEMPT	β	Preempt the specified resource.
QUEUE	γ	Hold entities in the specified file until a server becomes available.
SELECT	γ	Select entities from the specified queues and assign them to the specified server according to the specified selection rule.
TERMINATE	γ	Destroy entities and/or terminate the simulation.

Figure 2.1 List of Slam II network node types with description and classification; α for gate related, β for resource related, γ for general flow related

data collection related and declaration related. Initialization related statements handle the initialization of identifiers, data files, simulation study parameters, run parameters, and random number streams. Statements in this category are denoted by α in **Figure 2.2.**

Data collection statements, denoted by β in **Figure 2.2**, enable the collection of observation variables, time persistent data, as well as independent variable and dependent variable data for plots and tables. In addition debug and trace data can be collected.

Declaration related Control statements facilitate specification of aliases, program run time storage requirements, network definition blocks and state events for continuous models. Statements of this type are denoted by γ in **Figure 2.2.**

ARRAY	α	Initialize global array
CONTINUOUS	α	Initialization for continuous model
ENTRY	α	Initialization of storage file
EQUIVALENCE	γ	Alias Specification
GEN	α	Initialization of study
INITIALIZE	α	Initialization of storage files, variables, and parameters of experimentation
INTLC	α	Initialization of Slam II global variables
LIMITS	γ	Specification of operational limits

Figure 2.2 Description of Slam II control statement types with description and classification: α for initialization related, β for data collection related, γ for declaration related (Continued . . .)

MONTR	β	debug and trace statement
NETWORK	α	Network definition block
PRIORITY	α	Ranking priority for storage files
RECORD	β	Data collection of independent variable for plots
SEEDS	α	initialization of random number streams
SEVNT	α	Conditions for state event
STAT	β	Data collection for statistics based on observations
TIMST	β	Data collection for time persistant variables
VAR	β	Data collection of dependent variable for plots

Figure 2.2 Description of Slam II control statement types with description and classification: α for initialization related, β for data collection related, γ for declaration related

Fortran Subroutines

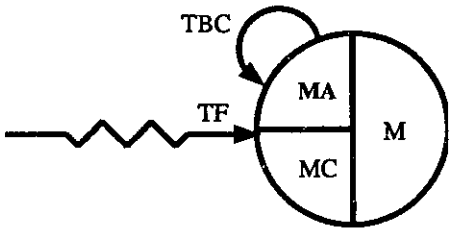
Fortran subroutines are used to increase the modelling flexibility of the language. Any calculations that cannot be handled using the base of network statements and control statements are coded as Fortran subroutines, which are called at appropriate times during the course of a simulation run.

2.3 Slam II Statement Format

Each Slam II network node has a corresponding network statement associated with it. A series of these statements taken as a group creates a functionally equivalent statement representation of the network node graph called the statement model.

CREATE,TBC,TF,MA,MC,M;

a)



b)

Figure 2.3 Example Slam II statements
a) Slam II CREATE statement
b) Slam II CREATE node

The network statements and control statements are powerful and are similar in form. Each statement is comprised of a statement keyword followed by a series of parameter fields delimited by commas. Many of these parameter fields can be left blank, in which case a default value is assumed for the parameter. **Figure 2.3 a)** shows the format of the CREATE network statement. Part **b)** shows the CREATE network symbol. **Figure 2.4** shows the format of the INIT control statement. Control statements have no symbols associated with them. The mnemonics following the statement keywords act as place holders for the parameter fields. To give the reader a feel for the syntactic structure of Slam II these statements are discussed briefly in turn below.

The CREATE statement is used to inject entities into the network. In the network model, it is the flow of entities through the network that drives the simulation. The CREATE statement can be supplied with parameter values to

INIT,TTBEG,TTFIN,JJCLR/NCCLR,JJVAR,JJFIL;

Figure 2.4 Slam II INIT statement

control characteristics regarding entity creations at the node. The TBC field is used for specifying the interval of time between entity creations occurring at the node. This parameter can be specified as a constant, variable or as a probability distribution. The TF parameter specifies the time at which the first entity creation is to occur in the course of a simulation run.

Each node residing in the network has a set of attributes associated with it. The attributes are stored in a one dimensional array called an attribute vector. An entity's attribute vector is named ATRIB and an entity's n'th attribute is referenced as ATRIB(n). The MA parameter specifies the index into the created entity's ATRIB vector at which the entity's creation time is to be stored (i.e., the entity's creation time is stored at ATRIB(MA)).

The MC parameter specifies an upper bound on the number of entity creations that could possibly occur at this node.

A CREATE node can have one or more arcs emanating from it, along which the created entities flow. The M parameter specifies the maximum number of replications that can be made of an entity created at the node.

The INIT statement, depending on how it is used, specifies either pre-study or pre-run activities for the simulation. The TTBEG and TTFIN parameters specify the start and stop times for the simulation respectively. The JJCLR and NCCLR, parameters are used to specify the statistics collection variables to be reinitialized

before each run commences. Statistics collection variables are used for storing statistical data collected during the simulation run. The JJVAR parameter is used to specify whether or not global variables are to be reinitialized prior to each run. And finally, the JJFIL parameter is used to specify whether or not the storage files are to be reinitialized prior to each run.

2.4 Slam II Program Structure

The skeletal structure of a typical Slam II program is shown in **Figure 2.5**. Pre-study activity is coded between the GEN statement and the NETWORK statement. Any Slam II control statement can be placed here. Typically, in this region one places statements that apply to all simulation runs. The Network model is placed between the NETWORK and ENDNETWORK statements. The topology of the network model is static and applies to all subsequently defined simulation runs, unless another network model is encountered further down in the program. Each SIMULATE statement initiates the execution of a simulation run. Pre-run activity is coded by placing the necessary Slam II Control statements before the SIMULATE statement.

2.5 Slam II Program Example

As an illustration of a Slam II program, consider the problem depicted in **Figure 2.6**. Here we have a warehouse to which locomotives arrive uniformly with a mean of 4 hours. The number of cars hauled by a train is exponentially distributed with a mean of 30. Cargo is unloaded from the train cars using fork lifts and each fork lift is assigned to a specific portion of the dock called a dock

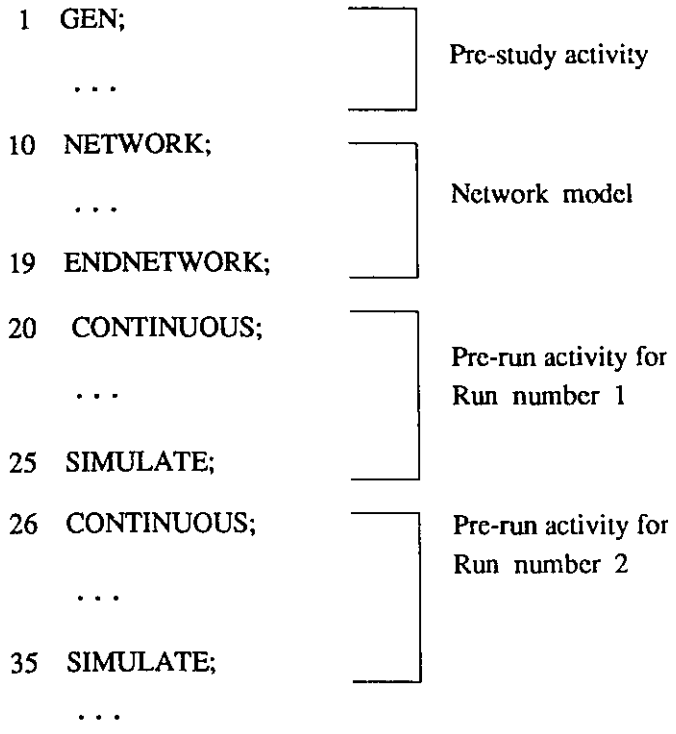


Figure 2.5 Skeleton of typical Slam II program

subarea. When a train arrives at the dock to have its cargo unloaded a subset of the total number of train cars is positioned in front of the loading dock at any one time, with each car positioned in front of a subarea. Each dock subarea has exactly one fork lift associated with it, and the fork lift within this subarea is responsible for unloading the train cars that arrive to its subarea.

The unloading operation for a single train car is uniformly distributed with a mean of 60 minutes and the dock supports the unloading of four train cars simultaneously. When a fork lift has finished unloading its assigned car it sits idle until the remaining fork lifts have finished unloading their assigned cars. Once all fork lifts have emptied their assigned cars the train moves forward slightly,

Loading Dock Problem

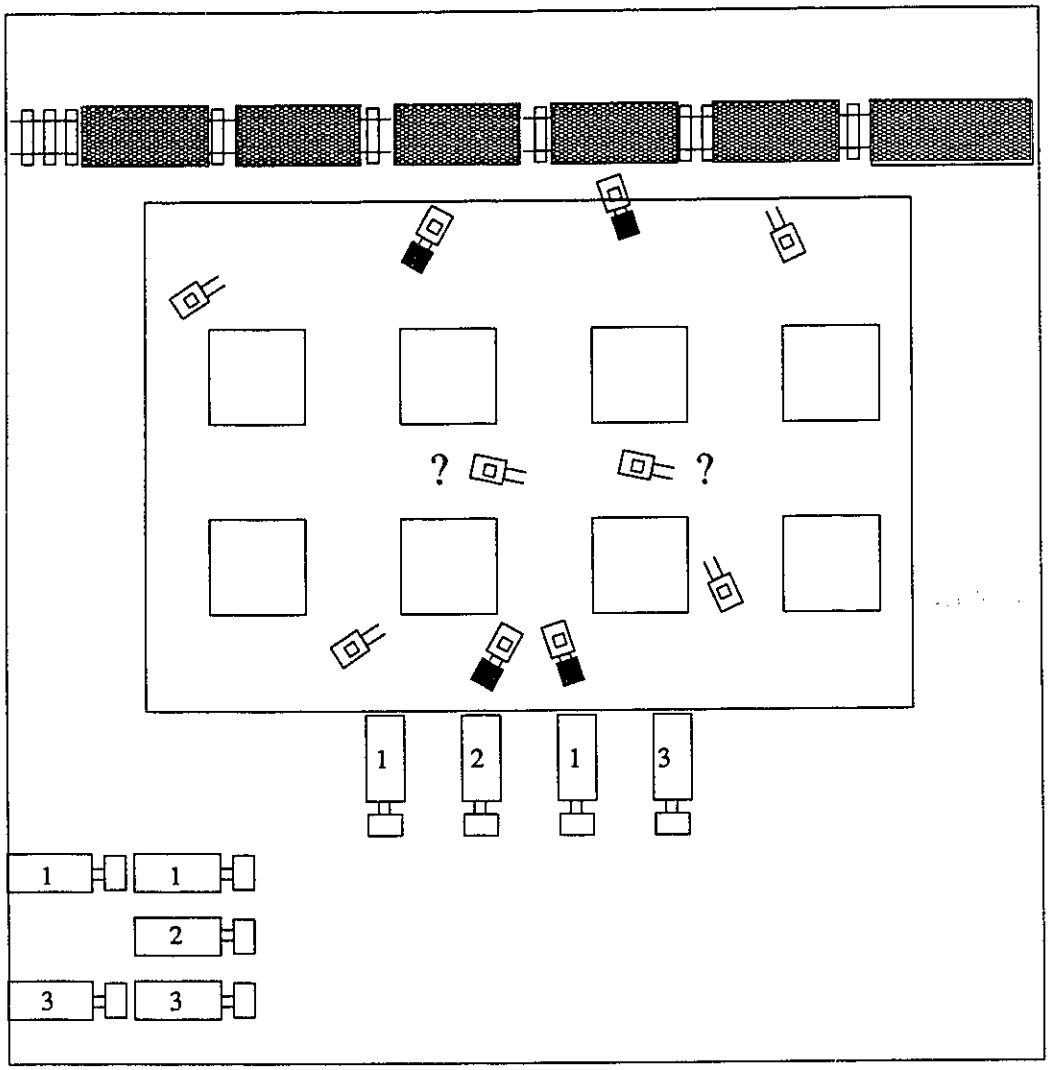


Figure 2.6 Schematic diagram of loading dock operation

bringing the next group of 4 cars in position for unloading. The duration of the train repositioning operation is 5 minutes. Any other train arrivals occurring while an unloading operation is already in progress are queued and served in a first come first served basis as the dock becomes available.

On the other side of the warehouse is a truck loading dock. Trucks belonging to three different carriers arrive at the warehouse. The frequency of arrivals is a function of the carrier. Trucks from carrier I arrive uniformly with a mean of 80 minutes. Trucks from carrier II arrive uniformly with a mean of 20 minutes, while those from carrier III arrive uniformly with a mean of 50 minutes.

All trucks are serviced by four fork lifts on a cyclical basis. If all truck fork lifts are in use, subsequent truck arrivals are queued. Trucks from carrier I are queued to a maximum of 4, trucks from carrier II are queued to an amount of 2, while those from carrier III are queued to an amount of 3. If the number of queued trucks exceeds the limit for that carrier, subsequent arrivals are immediately dispatched elsewhere. The duration of an unloading operation is uniformly distributed with a mean of 40, 30 and 60 minutes for carrier I, II, and III respectively. Furthermore, fork lifts assigned to train unloading operations are never used for truck loading operations, and vise-versa, even if they are idle.

The operators of the warehouse would like to purchase two additional fork lifts to service train unloading and truck loading operations They would like to know whether these new fork lifts should both be allocated to the train unloading side, to the truck loading side, or one forklift to each. A computerised simulation is desired in order to determine which scenario minimizes the sum of average train car waiting time and average truck waiting time.

The Slam II statement model for the solution to this simulation problem is shown in **Figure 2.7, 2.8 and 2.9**. This example will be referred to in subsequent chapters while discussing E/Slam's program elucidation ability and QA/Slam's quality assurance mechanisms.

```
;*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
;   MODEL OF LOADING DOCK OPERATION                .
;   XX(1) :  NUMBER OF TRAIN CARS LEFT TO UNLOAD.
;   XX(2) :  SUBAREA ALTERATION AMOUNT.
;   XX(3) :  USED IN FORKLIFT NUMBER ASSIGNMENT.
;   XX(4) :  NUMBER OF FORK LIFTS AVAILABLE FOR UNLOAD OPERATIONS.
;   XX(5) :  DURATION OF TRAIN REPOSITIONING OPERATION.
;*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
GEN,WENDT,LOADING DOCK,02/11/92,2,Y,,Y,Y,N,72;
LIMITS,8,5,100;
EQUIV/XX(5),REPOS/XX(1),TRAINCARS;
NETWORK;
      GATE/POSITION,CLOSE,3;
      RES/TNFORK(1),1;
      RES/DOCK,2;
      RES/4,TKFORK(4),4;
      RES/5,TKFORK2(6),4;
; TRAIN POSITIONING OPERATIONS:
;       ATR(1)     :     TRAIN ARRIVAL TIME.
;       ATR(2)     :     TRAIN LENGTH.
;
      CREATE,UNFRM(240),0,1,1;
      ASSIGN,ATRIB(2) = EXPON(30);
      ASSIGN,TRAINCARS = ATR(2),ATR(5) = 2;
      AWAIT(2),ATRIB(5);
      ASSIGN,XX(2) = XX(4) - NNRSC(TNFORK);
      ALTER,TNFORK/XX(2);
NXT   GOON,1;
      ACT,,TRAINCARS.LE.0,DPRT;
      ACT,,TRAINCARS.GE.XX(4),RDY;
      ACTIV,,TRAINCARS.GT.0.AND.TRAINCARS.LT.XX(4);
      ASSIGN,XX(2) = XX(1) - XX(4);
      ALTER,TNFORK,XX(2);
RDY   OPEN,POSITION;
      ACT,REL(T),,NXT;
;
; TRAIN PULLS AWAY:
;
DPRT  GOON;
      ACT,4;
      FREE,DOCK;
      COLCT(1),INT(1),TRAIN SVC TIME,8/15/15;
      TERM,20;
```

Figure 2.7 Slam II statement model of loading dock operation (Continued . . .)

```
; TRAIN UNLOADING  OPERATIONS:                                          44
;       ATR(1)    :     FORKLIFT NUMBER.                                45
;       ATR(2)    :     BATCHED ENTITY REFERENCE.                       46
;                                                                       47
      CREATE,0,0,,4;                                                    48
      ASSIGN,XX(3) = XX(3) + 1,ATR(1) = XX(3);                          49
NXTG  GOON;                                                             50
      AWAIT(3),POSITION;                                                51
      AWAIT(1),TNFORK,BALK(IDLE);                                       52
      ACT,UNFRM(60);                    UNLOAD OPERATIONS.              53
IDLE  BATCH,,4,,,ALL(2);                                                54
      CLOSE,POSITION;                                                   55
      ACT,XX(5);                        TRAIN MOVES FORWARD.            56
      UNBATCH,2;                                                        57
T     GOON;                                                             58
      ACT,,,NXTG;                                                       59
ENDNETWORK;                                                             60
;                                                                       61
;   TRUCK LOADING OPERATIONS:                                           62
;   ATR(2) :  TRUCK CARRIER NUMBER.                                     63
;   ATR(3) :  TRUCK ARRIVAL TIME.                                       64
;   ATR(4) :  TIME REQUIRED TO UNLOAD TRUCK.                            65
;   ATR(5) :  RESOURCE NUMBER TO SEIZE                                  66
;                                                                       67
;   FILE 1 :  QUE FOR CARRIER I                                         68
;        2 :   "         "    II                                       69
;        3 :   "         "    III                                      70
;        4 :  WAITING QUEUE FOR TKFORK                                  71
;        5 :  FULL QUEUE CARRIER I                                      72
;        6 :   "            "    II                                    73
;                                                                       74
      CREATE,UNFRM(80),,3;                 ARRIVALS OF CARRIER 1      75
      ASSIGN,ATR(2)=1,ATR(4)=UNFRM(40),ATR(5)=XX(4);                    76
QI    QUE(3),,4,BALK(FUL1),DISP;                                        77
      CREATE,UNFRM(20),,3;                 ARRIVALS OF CARRIER 2      78
      ASSIGN,ATR(2)=2,ATR(4)=UNFRM(30),ATR(5)=XX(4);                    79
QII   QUE(5),0,2,BALK(FUL2),DISP;                                       80
      CREATE,UNFRM(50),,3;                 ARRIVALS OF CARRIER 3      81
      ASSIGN,ATR(22)=3,ATR(4)=UNFRM(60),ATR(5)=XX(4);                   82
QIII  QUE(6),,3,BALK(EXT),DISP;                                         83
```

Figure 2.8 Slam II statement model of loading dock operation (Continued . . .)

```
DISP   SELECT,CYC,,QI,QII,QIII;                                    84
UNLD   AWA(4),ATR(5);                                              85
       ACT,ATR(4);                                                 86
       FREE,ATR(5);                                                87
       COLCT,INT(2),TRUCK SVC TIME;                                88
       TERM;                                                       89
FUL1   PREEMPT(7)/LOW(2),ATR(5),RPR1,,4;                           90
       ACT,,,UNLD;                                                 91
RPR1   ASSI,ATR(2)=0;                                              92
       ACT,,,QI;                                                   93
FUL2   PREEMPT(8)/LOW(2),ATR(5),RPR2,4;                            94
       ACT,,,UNLD;                                                 95
RPR2   ASSIGN,ATRIB(2)=0;                                          96
       ACT,,,QII;                                                  97
EXT    TERM;                                                       98
;                                                                  99
; XX(4) : RESOURCE NUMBER TO USE IN RUN                            100
;                                                                  101
MONTR,TRACE;                                                       102
REC,TNOW,TIME,P;                                                   103
VAR,NNQ(2),T,TRAIN QUEUE;                                          104
VAR,NNQ(3),1,CARRIER 1 Q LEN;                                      105
VAR,NNQ(5),2,CARRIER 2 Q LEN;                                      106
VAR,NNQ(6),3,CARRIER 3 Q LEN;                                      107
INIT,,5000,Y/1,,N;                                                 108
INTLC,XX(3)= 0,XX(4)=4,XX(5)=5;                                    109
SIM;                                                               110
INIT,,5000;                                                        111
INTLC,XX(3)=0,XX(4)=5;                                             112
SIM;                                                               113
INIT,100,5000;                                                     114
INTLC,XX(4)=4;              4 FORK LIFTS                           115
SIMULATE;                                                          116
INTLC,XX(4)=5;              6 FORK LIFTS                           117
SIM;                                                               118
```

Figure 2.9 Slam II statement model of loading dock operation

3 Architecture of E/Slam (Elucidation of Slam II Programs)

The following chapter provides an architectural description of E/Slam. The major components of the system are introduced to establish a framework for discussing program understanding, program elucidation and quality assurance issues in subsequent chapters.

3.1 E/Slam Components

E/Slam is a program understanding system comprised of an analysis phase, synthesis phase, update phase, and quality assurance phase (QA phase). Each phase consists of one or more modules that interact with an internal model of the Slam II program. The components from which E/Slam is constructed are given in **Figure 3.1**. Solid shadow boxes represent E/Slam input and output files. The hashed shadow box represents data abstraction, and boxes without shadows denote the major functional components of E/Slam. Directed arcs represent information flow.

3.1.1 Analysis Phase

In the analysis phase one is interested in studying an object to determine its elements and the relationship between these elements. The phase completes with a representation of its findings in an internally usable form.

E/Slam is in the analysis phase whenever it utilizes its source program analyzer. The source program analyzer is used to carry out a parse of the Slam II program. The module generates an internal model of the program, that is used

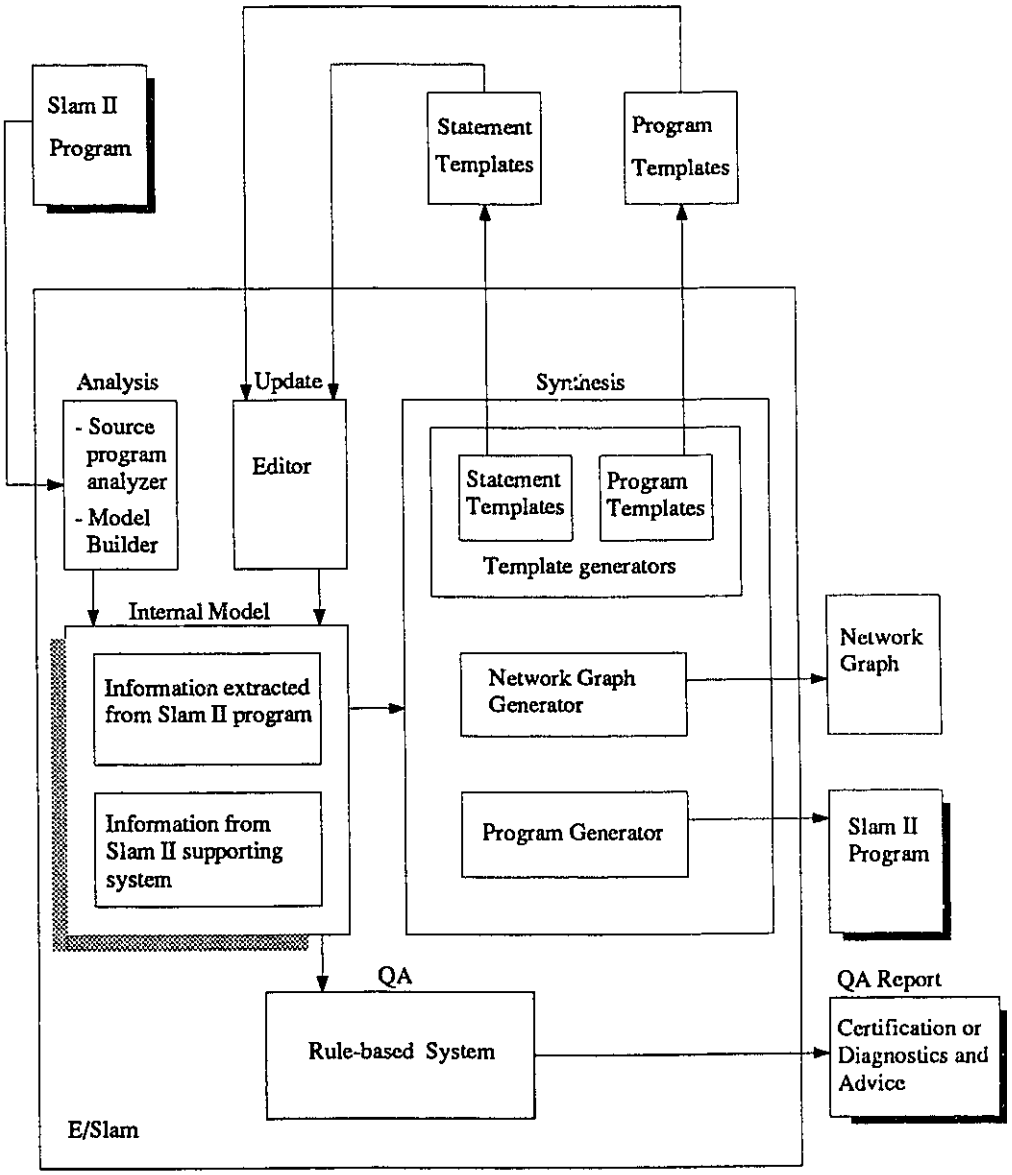


Figure 3.1 Architecture of E/Slam system.

as a basis for further study. This module is automatically entered when the user loads a Slam II program into the E/Slam environment. It is also entered when updates are made to the internal model through the template editor. The source program analyzer encompasses the Slam II parser and model builder. The model builder places parsed information in the appropriate data structures, making the information available to the synthesis phase and QA phase.

3.1.2 Internal Model

The internal model of E/Slam contains information about the Slam II supporting system (i.e., the Slam II language definition and its implementation; see Section 5.1.3), combined with information extracted from the Slam II program under study. The former is necessary for E/Slam to be able to interpret the Slam II syntax it encounters in the simulation program. Conceptually this component of information is part of the internal model. Physically it is distributed amongst the different modules in E/Slam.

The latter component of the model consists of information on the content of statements in the simulation program, as well as information on the ordering of statements in the program (program topology). The information held in this part of the model varies as a function of the Slam II simulation program under study.

3.1.3 Synthesis Phase

The synthesis phase is responsible for extracting information from the internal Slam II model and manifesting it in a form intelligible to the user. There are three modules that belong to this phase; each is distinguished according to how the information extracted from the internal model is elucidated.

The template generation module extracts information from the internal model and presents it to the user through two types of documentation templates: statement templates and program templates. The templates provide a framework for elucidating program information. Statement templates and program templates are distinguished according to how program information is grouped and presented. Statement templates elucidate program information on a statement-by-statement basis, while program templates elucidate information by taking a vertical slice of the program. The reader is referred to (O'Reilly, Nordlund 1989) for another template approach for documenting Slam II programs.

Template selections are made using pull down menus. Additional methods for template activation are also provided. Once a template is generated it remains on the screen until the user collapses it to an icon or removes it via a menu selection.

The network graph generation module uses the internal model of the Slam II program to generate a Slam II network model of the program. The network graph is built by taking the network statements in the program, mapping them onto their equivalent Slam II network symbols, and interconnecting these symbols according to the arrangement of statements in the program.

The program generator references the internal model and generates a Slam II program source code equivalent that reflects parametric changes made to the internal model. This mode is typically entered following a template edit activity. Output is saved in a file that can be directly input to a Slam II compiler.

3.1.4 Update Phase

The update Phase is responsible for facilitating and implementing user specified changes to the internal model of the Slam II program. Edits are made through documentation templates created by the template generator and are carried out by the template edit module. The template edit feature gives one the power to specify new simulation experiments on existing simulation programs.

3.1.5 QA Phase

The quality assurance (QA) phase references the internal model of the Slam II simulation program and analyzes it to identify inconsistencies in program specification. An inconsistency exists when a specification in one place contradicts a specification elsewhere in the Slam II program. An embedded knowledge-based system called QA/Slam carries out the consistency checking operation. QA/Slam is implemented in Prolog. A QA/Slam preprocessor converts the internal model of the Slam II program into Prolog facts that are entered into a fact base which serves as input to QA/Slam. If inconsistencies in program specification are found, the user is notified of the problem and is given suggestions about how to fix it. If no inconsistencies are found then QA/Slam provides a certification of the program's correctness.

3.1.6 Program Generation Phase

The program generation phase is responsible for producing an up to date source code version of the content of the internal model. The modules in this phase extract program information from the internal model and produce a compilable version of the Slam II program. Program changes introduced during the update

phase are reflected in the newly generated version of the Slam II program (i.e., the program source code).

3.2 Order of Activation of the Modules

With the exclusion of source program analysis, the user has the power to activate any of the phases directly through the user interface. This means the order in which modules are activated is under user control. Modules can be entered repeatedly and in different orders according to the user's purpose for using E/Slam. Nevertheless, there is a logical order in which the modules are intended to be used and this is discussed in the following paragraphs.

Once the user enters the E/Slam environment, the next action is to load a Slam II program. This is initiated through one of the pull down menu options. At program load time, E/Slam enters source program analysis mode to build an internal model of the Slam II program.

Once the internal model is in place, other modes of operation can be entered. The subsequent modes entered are dictated by what the user wants to do with the Slam II program. If the user's goal is to use E/Slam as an aid in understanding the Slam II program, then the user would initiate the network graph generation mode or the template generation mode, in order to display documentation information about the Slam II program. In this scenario the user does not have a need for the template edit and program regeneration modes. If the user's goal is to understand the simulation program and then modify it to carry out new simulation experiments then these additional modes would be used.

If one wishes to modify existing experiments in order to specify new experiments, this is done by entering template edit mode. This mode can only be entered following template generation mode. To initiate a change, the user highlights the appropriate value in the dynamic area of the documentation template and then uses the menu to select edit mode. A window pops up to enable the user to enter a new value for the highlighted field. Once a value is entered, a syntactic analysis is carried out on the newly specified value with respect to the context in which it will appear in the program. If the syntax is incorrect E/Slam does not accept the new value: the edit window remains open until a proper value is specified or the edit operation is cancelled by the user. If the syntax is correct, the internal model is updated in the appropriate place to reflect the change. The template edit mode is discussed further in Chapter 7.

Following any program edits, and prior to initiating program regeneration, QA/Slam is used to ensure user initiated program changes have not caused the simulation program to become inconsistent. This module is initiated by the user via a panel button in E/Slam's user interface. Following the consistency checking operation, a text window pops up to display the QA report generated by QA/Slam. If the QA report identifies program inconsistencies, then the user enters template edit mode to correct the problem. In addition to identifying inconsistencies, the QA report offers suggestions on how to eliminate the inconsistent states. After making the necessary corrections, the user enters the QA phase again to ensure the change is acceptable. QA/Slam is discussed in detail in Chapter 8.

Another way of using E/Slam is to immediately activate QA/Slam when the Slam II program is loaded into the environment to determine if any inconsistencies

exist. If no inconsistencies are found, then the designer has a certification of its correctness.

Once all inconsistencies are identified and corrected, the user initiates program generation mode to obtain a compilable copy of the modified program. program generation mode is discussed in detail in Chapter 7.

4 E/Slam's Internal Model

The internal model of E/Slam consists of two primary data structures: a statement list structure, and a symbol table. The statement list structure contains complete information about each statement occurrence in the Slam II program. The symbol table structure records usage information for each program element. These structures are discussed in detail below.

4.1 Statement List Structure

Statement information is grouped by statement type, in a data object called Stmt_Type_Indx_List[]. This vector contains 43 cells: one for each Slam II statement type. Information on each occurrence of a given statement type is stored in a linked list whose head is pointed to by the appropriate cell in Stmt_Type_Indx_List[]. The list head pointers are ordered alphabetically, thus Stmt_Type_Indx_List[0] dereferences the linked list information on all ACCUMULATE statements in the Slam II program while Stmt_Type_Indx_List[42] points to information on all VAR statement occurrences.

In the linked list of statement information there are three main types of nodes: statement nodes, field nodes and token nodes.

4.1.1 Statement Nodes

Statement nodes contain useful information that applies to the Slam II statement as a whole. As seen in Figure 4.1 a **Stmt** node consists of nine members. Each member is described in the following paragraphs.

```
typedef stmt
   {
     Statement_type id;
     char           *name;
     int            name_order;
     int            stmt_num;
     char           stmt_label[MAX_LABEL_SIZE];
     int            label_order;
     list           whitespace;
     list           field_list_head;
   } Stmt_rec, *Stmt;
```

Figure 4.1 Typedef for an E/Slam **Stmt** node

The *id* member identifies the type of Slam II statement being documented (i.e., CREATE, INIT, etc.). *Statement_type* is an enumeration of all statement types in the Slam II language.

The *name* member contains the character string representation of the statement keyword as it appears in the program. Since only the first three letters of a statement keyword are necessary to identify the statement, Slam II programmers do not always specify the full keyword. E/Slam must keep track of how many letters were actually specified on a per-statement basis for when it comes time to regenerate the program listing from the internal model.

When a Slam II statement is parsed, it is broken up into tokens. To enable the statement to be reconstructed during the program generation phase, it is necessary to maintain an order on the tokens identified. This is done by associating an integer value with each token. When it comes time to reconstruct a statement field, the tokens are appended to each other according to increasing order number. The *name_order* member is used to record the order number for the Slam II statement keyword.

The statement number member *stmt_num*, records the line number at which the statement occurs in the program. This information is used to identify the scope of each statement (i.e., which simulation run it belongs to). It is also used in reconstructing the program during the generation phase.

The statement label member *stmt_label* contains the statement label associated with the Slam II statement. In the Slam II program, the statement label resides to the left of the statement keyword.

The *lable_order* member is used to record the order number for the Slam II statement label so that when it comes time to reconstruct the statement, its label appears in the correct position relative to the other statement tokens.

The *whitespace* member is a pointer to a linked list of nodes. Each node contains a text string of the characters that lie between two parameter fields of a Slam II statement. This includes all blank space and field delimiters. It is necessary to retain such information, because when it comes time to reconstruct the statement, we want the statement format to appear exactly as it did prior to parsing. Each node in the linked list has an order number associated with it. This is used to ensure the white space and parameter field values are reassembled in the proper sequence. The *list* typedef is a two member structure. One member points to the data associated with the list node, while the other element points to the successor in the linked list.

The *field_list_head* member is a pointer to a linked list of **Field** nodes, where each **Field** node contains information on a single field in the Slam II statement.

4.1.2 Field Nodes

A **Field** node is used to record information on a single Slam II statement field. As seen in Fig 4.2, this structure consists of three members.

```
typedef struct field
   {
     int              field_name;
     Boolean          defaulted;
     list             parse_list_head;
   } Field_rec, *Field;
```

Figure 4.2 A Typedef for an E/Slam **Field** node

In Slam II each statement field has a name which is used to distinguish it from other fields in the same statement. This field name is held in the *field_name* member of the **Field** structure.

In Slam II, statement fields can be defaulted by the programmer. This occurrence is reflected in the boolean member *defaulted.* If the value of this field is *true* then the program statement field has been left blank, otherwise, the field contains an explicit value. Explicit and default values are discussed in Chapter 5.

The third member named *parse_list_head*, is a pointer to a linked list of token nodes that comprise the field's value.

4.1.3 Token Nodes

A token node is a holding place for a Slam II language token. The tokens comprising a Slam II statement field value are strung together in linked list fashion.

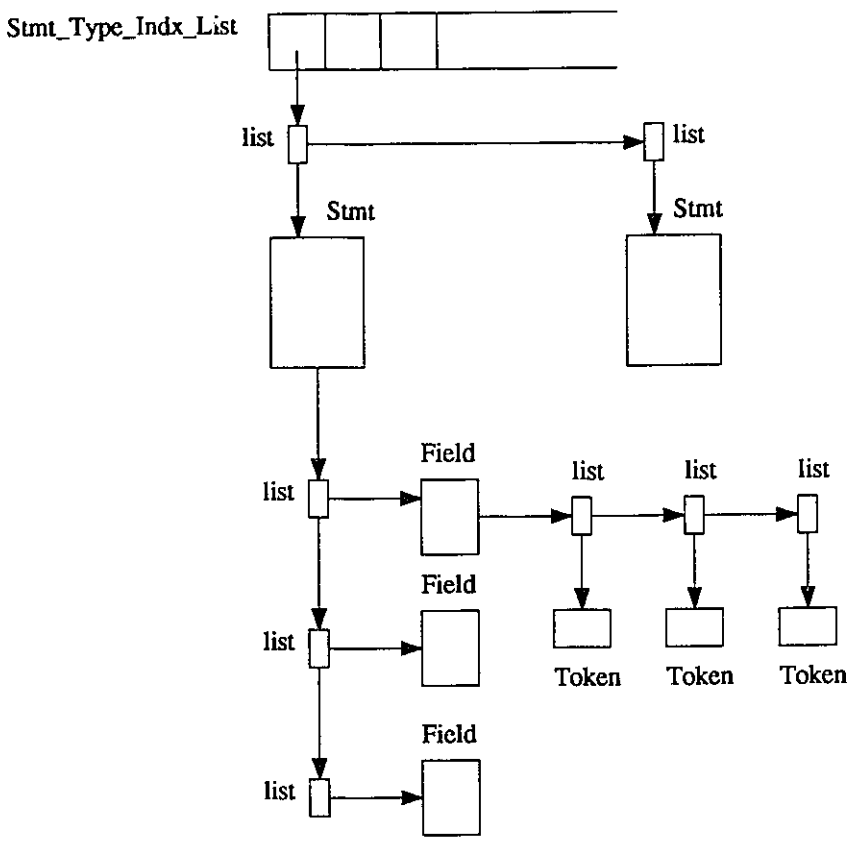


Figure 4.3 Skeletal layout showing the interconnections between E/Slam's internal data structures

This list is pointed to by the *parse_list_head* member of the associated **Field** node. Breaking a statement field value into its constituent tokens makes it possible for E/Slam to validate field values. It also enables E/Slam to assign new values to fields and for QA/Slam to carry out consistency checks on fields. **Figure 4.3** gives a typical representation of how the above mentioned structures link together to describe a Slam II program.

4.2 Symbol Table

The symbol table is used to record frequently accessed data on resources, gates, files and system variables. In Slam II a one to one correspondence exists between resource name and resource number. The resource number is usually assigned implicitly by Slam II. It is often necessary to know the resource number associated with a particular resource name and vise versa. This mapping is stored in the symbol table for quick access. This procedure is also followed for Slam II gates. The symbol table also holds information about which file numbers are assigned to which resources and gates in the Slam II program.

The symbol table holds information on cross references between program statement fields and template fields. This information is drawn upon during the template edit phase. When the user specifies a new value for a template field, the symbol table is referenced to determine the location in the Stmt structure where that value was extracted. Once the information source is identified the new value can be inserted into the internal model.

5 Program Understanding and Slam II

"What we do not understand we do not possess"—Goethe

The human ability to "understand" that which we experience has been a philosophical topic of discussion since the time of Socrates. Much of the insight offered by philosophers of the past is just as applicable when talking of computer-based understanding systems of today. For instance, the view that understanding is a model-based activity applies equally to AI applications with understanding ability, as it does to human understanding. The following is a refinement of a definition of understanding given by Ören (1992):

A system S_1 can understand a system S_2, if

1) S_1 has a class model C, of S_2,
2) S_1 can analyze S_2 to find its elements and their relations, to produce an instance model C' , and
3) S_1 can establish links between the elements of C' as well as relationships between S_2 and C'.

The class model C is a generic model that acts as the infrastructure for establishing the instance model C'. It includes the a priori knowledge required to establish elements and links between elements. The instance model describes the elements and relationships regarding system S_2. The distinction between class

model and instance model is analogous to the concept of class and instance of a class in object oriented programming.

The term "system" as it is used in the above definition can apply equally well to a human as it does to a computer based system. If system S_1 is a human then we have a definition of human understanding. If we are talking about a computer-based program understanding system then S_2 is a computer program and S_1 is a program understanding system. Examples of program understanding systems are E/Slam, as well as KAESTLE, FOOSCAPE, and TRISTAN (Böcker 1991).

Once a mapping between C' and S_2 is established, system S_1 can perform other knowledge processing activities on C'. For program understanding systems this additional knowledge processing may involve:

1. Offering help and guidance to the programmer.
2. Assisting in program testing, verification and debugging.
3. Offering assistance during maintenance, revision and updates
4. Assisting the programmer in understanding what a program is doing, and how it does it.

As the reader will see in Chapter 8, the knowledge processing abilities listed in items 1 and 2, correspond to the features of QA/Slam. QA/Slam offers help and guidance to those making updates to Slam II programs. It offers assistance in program verification by performing consistency checks on programs, to certify their correctness. E/Slam offers assistance in making program updates and provides assistance to those who wish to understand Slam II programs. Thus, E/Slam belongs to category 3 and 4.

The degree and sophistication of knowledge processing ability is a function of the granularity of the instance model C' (i.e., how many elements and relationships have been identified). Ning (1992) identifies four levels of program understanding: text level, syntactic level, semantic level and concept level. Text-level understanding involves recognizing the program as a series of text characters. This is the lowest level of understanding. Syntactic-level understanding involves recognizing the language constructs in the program. Understanding is at the statement level, which means constructs such as IF statements, and FOR statements are recognizable. To reach this level a parse of the program is required. Semantic-level understanding involves deriving semantic information about programs. This includes control flow, data flow, etc. Concept-level involves understanding a program from a problem-oriented perspective. Here, meaning is described in terms of the real would problem being solved.

E/Slam's ability reaches to the semantic-level. In order to reach this level of understanding, the Slam II information sources that must be tapped include program statement fields, program topology and Slam II supporting system.

5.1 Sources and Types of Information For Understanding Slam II Programs

As shown in **Figure 5.1** there are three sources of information that must be tapped for E/Slam to understand Slam II programs. Information must be collected from parameter fields within the program statements, from the program topology, and from the supporting system. Each information source contributes different kinds of information to the program understanding effort.

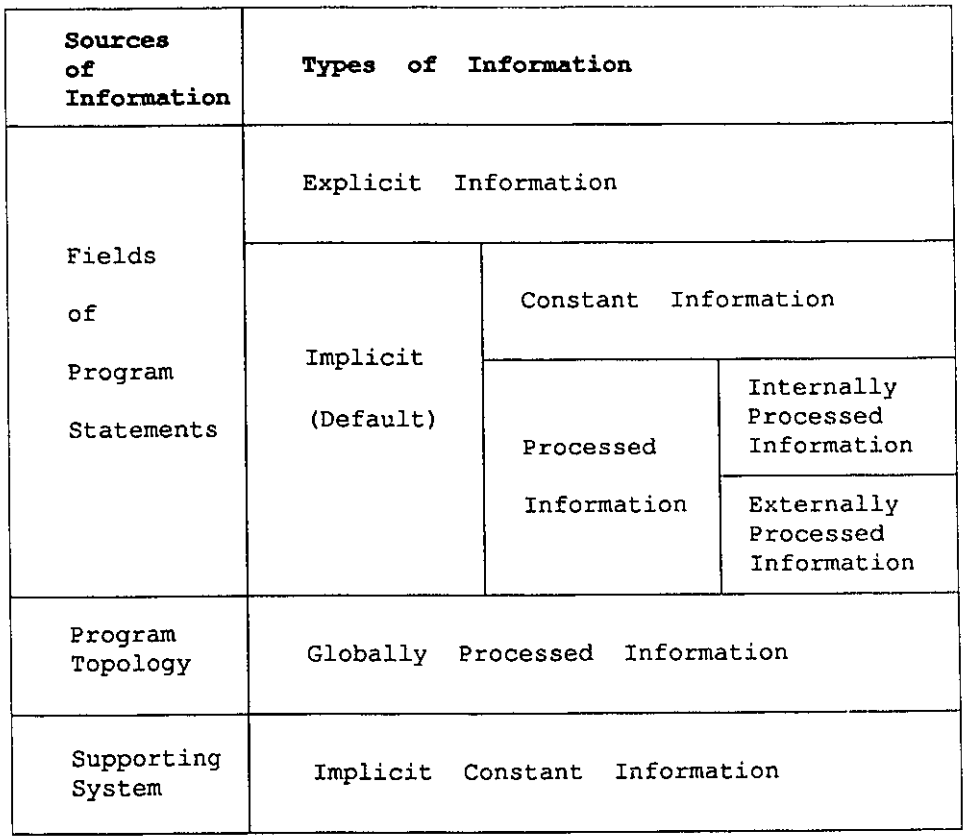


Figure 5.1 Sources and types of information for E/Slam program understanding system

5.1.1 Information Extracted From Fields of Statements

Program statement fields may contain explicit information or implicit information. Explicit information exists when an actual value is present in the parameter field. This value is called an explicit value. In contrast, when a parameter field is left blank it is said to contain implicit information, and the value assigned by the language is termed an implicit value, an implied value, or a default value.

Figure 5.2 gives an example of how both explicit and implicit information are associated with a typical Slam II INIT statement. In this example, explicit information is present in the TTFIN, JJCLR, NCCLR and JJFIL fields. The explicit value in the TTFIN field is 77. The explicit value in the JJCLR , NCCLR and JJFIL fields are Y, 1 and N respectively. Implicit information is associated with the TTBEG and JJVAR fields since no values have been entered in these locations. According to the definition of the INIT statement, Slam II assigns an implicit value of 0 to the TTBEG field and an implicit value of 'Y' to the JJVAR field. If the JJFIL field had been left blank, it would have been assigned an implicit value of 'Y' as well.

Implicit information can be further distinguished according to the method the language uses for assigning a value to the field. If the language always assigns a constant value then this is called constant information and the value obtained is

Statement syntax	INIT, TTBEG , TTFIN , JJCLR / NCCLR , JJVAR , JJFIL;					
Program statement values		77	Y	1		N
Type of information	implicit				implicit	
		explicit	explicit	explicit		explicit
Default values when information is implicit	0	infinity	Y	all	Y	Y

Figure 5.2 Explicit and implicit information in a Slam II INIT statement

called a default constant. In the previous example default constants were assigned to the TTBEG field and JJVAR fields.

In contrast to default constant value assignment, some statement fields require some degree of processing in order to determine their implied values. This situation arises when an implied field value is a function of values specified elsewhere in the program. In this scenario, we are dealing with processed information and the value obtained is called a processed value.

Processed information can be distinguished based upon the scope of processing necessary to determine the implicit value for a statement field. If it is only a matter of referencing parameter values specified elsewhere within the same statement in order to determine the value for the field, then the information obtained is called internally processed information. If it is necessary to go outside the scope of the current statement to determine the value for a field then the implicit information is termed externally processed information.

The nature of processing necessary to determine the value of a field varies with each field. In some cases it is a matter of referencing the value of another field and using that same value. In other cases additional calculations, usually numeric, are involved. To clarify the distinguishing features of each definition, consider the following examples.

As seen in **Figure 5.3**, the DTMIN field of the Slam II CONTINUOUS statement is a good example of internally processed information. The DTMIN field is used to specify the minimum step size to be used by the integration algorithm in continuous simulations. If this field is defaulted, the value assigned by Slam II is a fraction of the value in the maximum step size parameter field

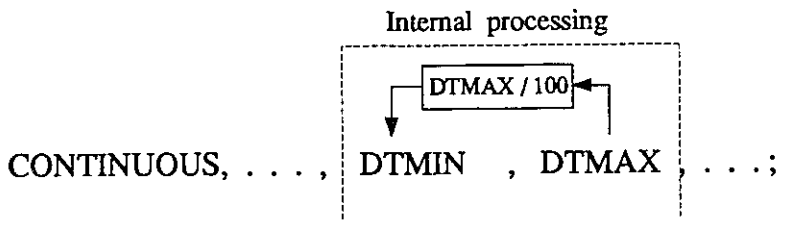


Figure 5.3 An example of internally processed information in a Slam II CONTINUOUS statement

(DTMAX). If DTMAX is also defaulted, Slam II applies the necessary algorithm for determining DTMAX's value. Whatever value is obtained here is then used in the algorithm that calculates the value of DTMIN.

For an example of externally processed information consider the Slam II RECORD statement shown in **Figure 5.4.** This statement has a field for specifying the time at which data recording is to commence (TTSRT). If this field is left unspecified, Slam II uses the simulation start time (TTBEG) specified in the INIT statement as the value for TTSRT. If TTBEG is also unspecified, Slam II does the processing necessary to obtain its value and assigns this value to TTSRT.

In the majority of cases the implied information associated with a Slam II statement field is default constant information. Nevertheless, it is necessary for a Slam II understanding system to have the ability to apply the necessary algorithms in order to obtain values for fields that involve processed information.

5.1.2 Information Extracted From Program Topology

Program topology, as a source of information, is treated separately from

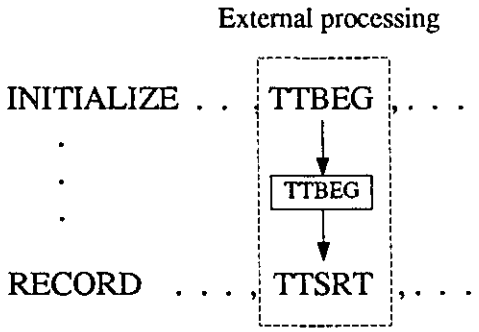


Figure 5.4 An example of externally processed information in the Slam II RECORD statement

information obtained through program statement fields, because the two differ in terms of focus. Information extracted from program topology involves studying the order of statements in the program and recognizing relationships built as a result of this ordering. It is not concerned so much with the contents of statement parameter fields as it is with a global view of the program layout.

The type of information obtainable from this source is termed globally processed information. It is global because it involves viewing the program as a whole. it is not possible to point to a single statement or field to extract the information. The information must be obtained through an analysis of the program.

In order to determine the run number with which a given control statement is associated, one must know the statement type, its position relative to other statements of the same type and its position relative to the SIMULATE statements in the program. These criteria involve a global perspective of the program. The information required is not contained in any one statement, it is contained in the

arrangement of statements in the program. It is this positioning of statements that must be studied in order to obtain a statement's run number.

As an example of this, consider the Slam II program fragment shown in **Figure 5.5**. Because the CONTINUOUS statement on line 20 is positioned before the first SIMULATE statement on line 25, the CONTINUOUS statement is associated with simulation run number 1. In addition to this, because another CONTINUOUS statement exists at line 26, before the second occurrence of the SIMULATE statement, the statement at line 26 applies to run number 2, overriding the parameter values previously specified in the CONTINUOUS statement at line 20. If the CONTINUOUS statement at line 26 had been removed, then the CONTINUOUS statement at line 20 would have applied to both run number 1 and run number 2, imposing the same parameter values for each run.

This example shows overriding at the statement level, where all parameter values in the statement override parameter values specified in a previous occurrence of the statement. Overriding can also occur at the sub-statement level, where subsequent occurrences of a statement override a subset of the values specified in a previous occurrence of the statement. The INTLC statement falls into this category. This issue raises the degree of complexity involved with program understanding.

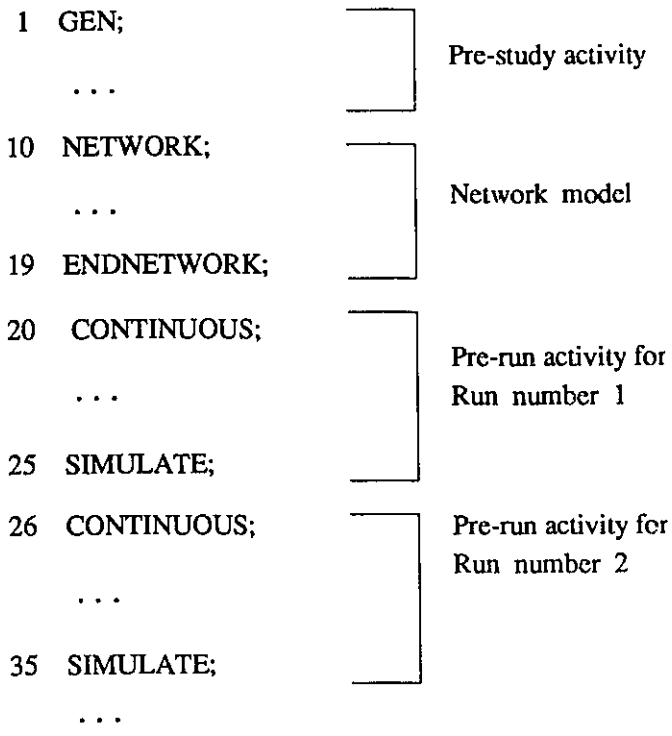


Figure 5.5 A Slam II program fragment showing how simulation run information is contained in statement positioning

5.1.3 Information Extracted From Supporting System

The third source of information useful to program understanding comes from the supporting system. The supporting system is comprised of the language definition and the implementation of the language definition. All information that falls into this category is implicit because it is either hidden in the language definition or the language implementation. This kind of information cannot be seen by observing the program listing. Because there is no way for one to change the information in this category, it is default information. Furthermore, it is constant information because it invariably applies to all programs.

An example of implicit information provided by the Slam II supporting system is the integration algorithm used for continuous simulation. In Slam II the state trajectories of continuous systems are calculated by a Runge-Kutta-Fehlberg (RKF) algorithm with variable step size. This algorithm is provided by the supporting system. It is constant information because it applies to all continuous simulations. It is default information because it is automatically applied, and it is implicit because nowhere in the program is this information stated.

5.2 Slam II Statements Represented as Finite State Machines

Slam II allows for a high degree of implicit specification. To be able to extract the implicit as well as explicit information from a Slam II program, we need a systematic representation of their structure. Doing this, provides a clear understanding of each statement, and ensures a more complete understanding than the existing textual descriptions of each statement's functionality. This section discusses finite state machines and how they can be used in systematizing the representation of Slam II statements.

A deterministic finite state machine (FSM) is a mathematical model that consists of a five-tuple (S,Σ,δ,S_0,F) where

- S is a finite set of states,
- Σ is a finite set of input symbols,
- δ is a transition function that maps state-input pairs to sets of states,
- $S_0 \in S$ is distinguished as the initial state, and
- $F \subset S$ is distinguished as a set of accepting states.

where for each state s ϵ S, there is at most one successor state that can be reached from s, given an input a ϵ Σ, (Aho et al. 1986).

An FSM can be represented diagrammatically by a transition graph. A transition graph is a directed graph in which states are represented as nodes and the transition function is represented by labelled arcs. An arc represents the state transition that would occur upon receiving an input that matches the arc's label. The node from which the arc emanates is viewed as the current state, while the node at which the arc terminates is the next state.

FSM's are valuable for specifying the valid syntax of programming language statements, especially when represented in the form of transition graphs. The visual nature of transition graphs allow one to clearly see the structure of the statement, and it provides a good model for thinking about statements. Implicit information is made explicit, as the transition graph shows the conditions under which statement defaults come into affect. Furthermore, because of its ability to provide a good thinking model, it promotes completeness in statement specification. These attributes make it a good intermediary step in the development of a parser.

E/Slam has a built-in parser for the Slam II language. Because Slam II consists of an extensive syntax of 43 program statements (excluding statements from the Material Handling Extension), each with a complexity similar to the statements in **Figure 2.3** and **Figure 2.4**, a finite state machine representation of each statement, expressed in the form of transition graphs proves advantageous as a precursor to parser development for this language. The nature of the specification allows for a much easier transition at implementation time.

The FSM's provide an unambiguous specification of how each statement is parsed and the conditions under which default values are assigned to parameter fields. This yields a parser implementation that instills a higher degree of confidence of correctness in the minds of its designers.

Figure 5.6 and **Figure 5.7** show finite state machine representations for the CREATE and INIT statements respectively. In the figures, states are displayed as rectangular boxes and input symbols are expressed as arc labels. The starting state is always the top most box and it contains the keyword name of the statement the FSM is defining. There is always only one final state and it is denoted by a double boarder box.

At the top of each Figure is a parametric representation of the Slam II statement. The mnemonics used to identify statement fields are identical to those used in (Pritsker 1986) and (O'Reilly, Lilegdon 1987). Descending along the left column is a textual description of the parameter being scanned. To the right of the textual description is the portion of the FSM responsible for identifying the parameter described.

One can see that the states alternate from value assignment states to acceptance states. Value assignment states are those containing an expression of the form "GET <mnemonic>", or an assignment expression of the form "<mnemonic> = <value>". It may seem unusual to have a phrase specifying an action as a label for a state. However, it is to be interpreted simply as an acknowledgment that the value specified is to be associated with the "<mnemonic>" for that field. In the former case, it is the association of whatever value exists in the statement field. In the latter case, it is the association of the field's default value "<value>".

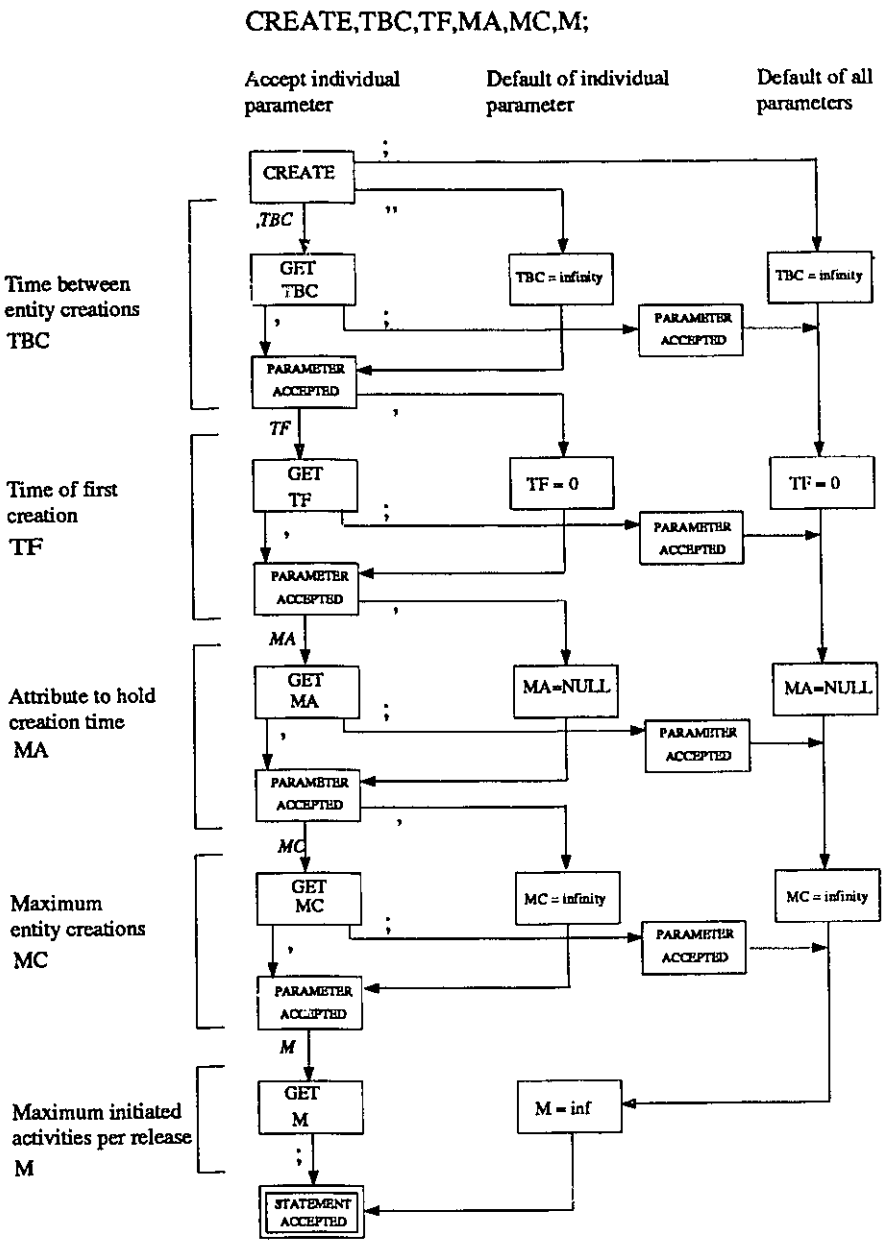


Figure 5.6 Finite state machine for the Slam II CREATE statement

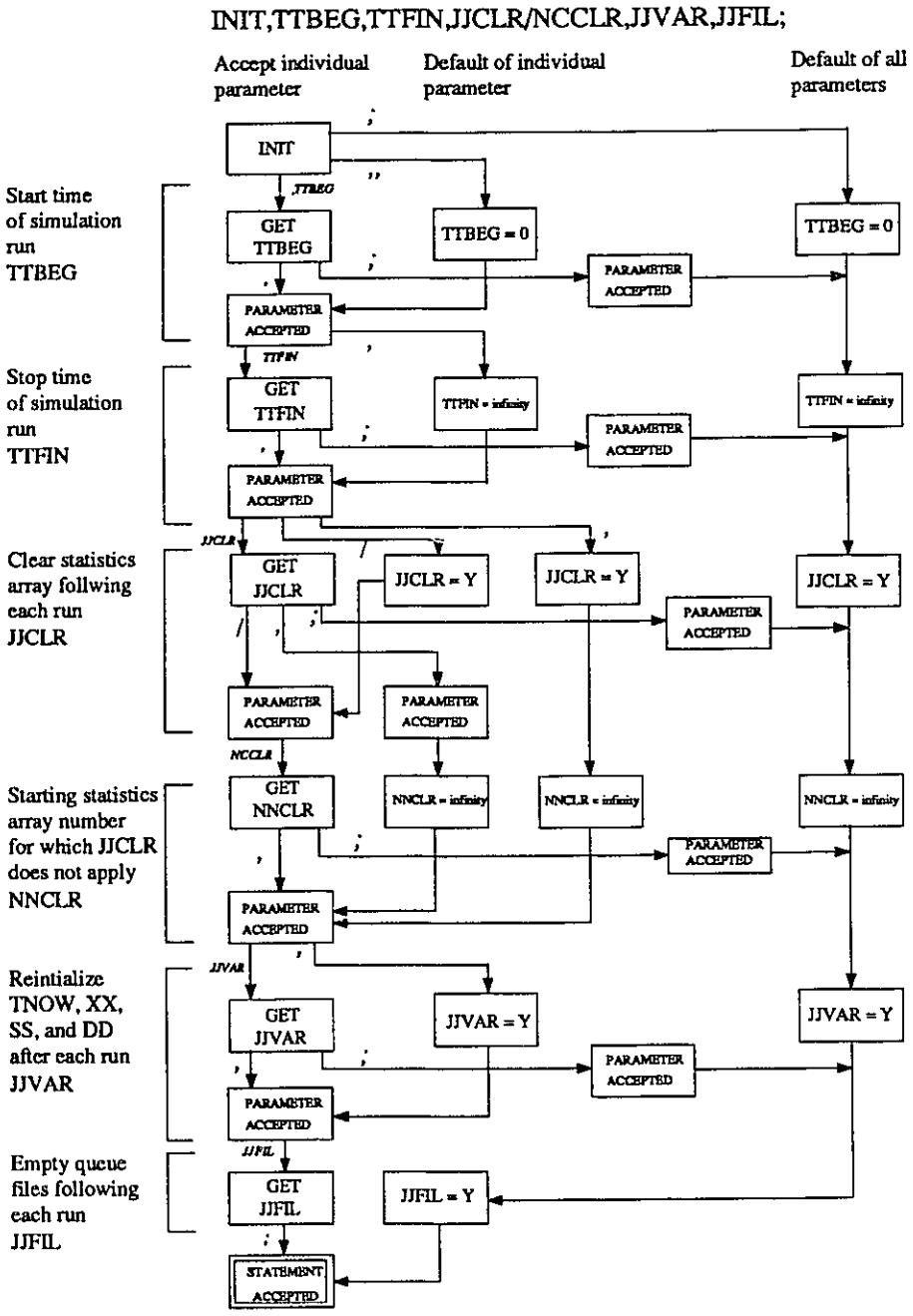


Figure 5.7 Finite state machine for the Slam II INIT statement.

It should be noted here that when a mnemonic name exists on an arc, an assumption is made that the string represented by the mnemonic is not a null string. For if this assumption were not made the FSM would be nondeterministic. Also, when a mnemonic name exists on an arc it acts as a substitution for the entire value specified in its associated parameter field.

The Acceptance states are those containing either the text "Parameter Accepted" or the text "Statement Accepted". The former type of acceptance state simply defines a boundary between parameter fields. This state shows that a successful parse has been made on the previous statement field and the FSM is about to parse the next field. The latter type of acceptance simply shows that a successful parse has been made of the entire statement.

Each FSM has its states arranged in columns and rows. For each parameter field, a row of states exists in the FSM and this row is found beside the textual description of the parameter field. The left most state in a row always represents the state obtained when an explicit value has been specified for the corresponding statement field. All other states to the right of it are obtained when the statement field has been defaulted.

Following the sequence of transitions from the starting state down the left most column constitutes the parse of a statement for which all parameter fields have been assigned explicit values. If a default value exists in a statement field then either a temporary or permanent deviation from the left hand column takes place. A permanent deviation takes place if the statement terminator character ";" has been scanned. Otherwise, a temporary deviation takes place.

To avoid making the transition diagrams too cumbersome to read, error states and transitions to error states have been omitted. If no transition can be taken, in response to a string of characters, then the rest of the input string is immediately discarded. FSM specifications have been created for all of the Slam II statements and can be found in (Wendt et al. 1989).

6 The Template Approach to Understanding Slam II Programs

"Even for the scientist the description in plain language will be a criterion of the degree of understanding that has been reached"—Heisenberg

Information extracted from the Slam II program and the Slam II supporting system is presented to the user through statement templates and program templates. This chapter begins by discussing a frame work for studying these two kinds of documentation templates. The framework will be used as a vehicle for presenting some sample documentation templates, and to demonstrate how the content of a template field relates to its information sources. All sample templates shown in this chapter have been created in reference to the Slam II example program discussed in Section 2.5.

6.1 Template Structure

E/Slam displays information extracted from a Slam II program through a set of documentation templates. Documentation templates present relevant program information to the user in a clear, consistent format, that eliminates unnecessary detail and brings implicit program information to the foreground. Two categories of documentation templates exist in E/Slam: statement templates and program templates. Each approach offers a view of the simulation program from a

perspective orthogonal to the other. Statement templates provide documentation on a statement-by-statement basis, while program templates take a vertical slice of the program (Ören, Zeigler 1979), organizing information according to predefined categories so that information can be presented to the user in a more abstract way.

The distinguishing characteristics of program and statement templates are discussed further in the following sections. However, before discussing the differences between the two forms of documentation, it is worth discussing the template structure they share in common. The template structure discussed in the following paragraphs provides the frame work for presenting program and statement information. It is what resides within this framework that distinguishes program templates and statement templates.

An E/Slam documentation template consists of two parts: a static part and a dynamic part as shown in **Figure 6.1.** The static part is comprised of a group of headings arranged in a columnar format. Each column contains textual information describing some aspect of the program under study. These column headings can be arranged in a stacked format where one column heading spans a group of sub-columns. In this format the spanning column contains general heading information and each sub-column spanned contains more specific heading information. In other words, the spanning column defines a domain and the spanned sub-columns partition that domain. This domain splitting technique can be applied to more than two levels.

When the user reads a domain heading and follows through to the spanned column(s) beneath, a documentation heading phrase is constructed that points out

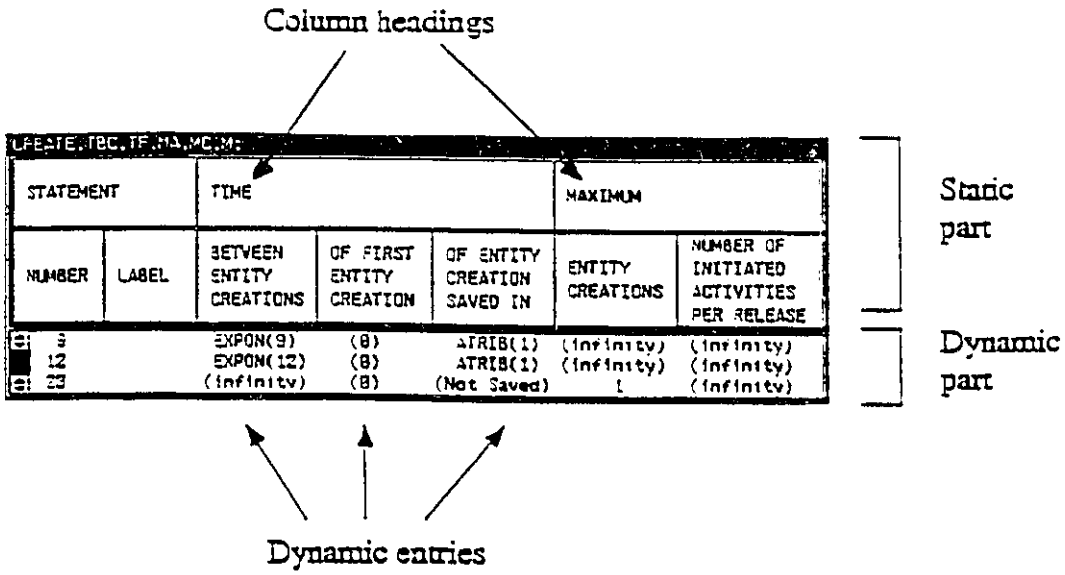


Figure 6.1 E/Slam documentation template structure

some important concept regarding either simulation studies in general, or Slam II programs in general.

Immediately below this column, values exist that bind the general concept to the simulation program currently under analysis in the E/Slam environment. If the value displayed in the template is enclosed in parenthesis, it means the primary information source from which the value was obtained contained an implicit specification. The region of the template that contains information extracted from the program under analysis is called the dynamic part of the documentation

template, simply because the values in the region vary as a function of the program under analysis. The portion of the template where the column headings reside is called the static part, because the textual information in this area remains unchanged regardless of the program being studied. Immediately above the static area of each statement template the syntax of the statement with which the template corresponds is given.

Documentation templates are constructed in a columnar format, as opposed to row format for specific reasons. By having the static part of the template positioned along the top of the window, and the dynamic part directly beneath, all instances of a statement type can be uniformly displayed within a single template, with each occurrence on a separate line in the dynamic area. The advantage of this, is a reduction in the number of windows on the screen. Due to the large number of Slam II statements—and thus a large number of template types—the number of templates that can be open simultaneously without crowding the workstation screen is an important issue. If the user wishes to see documentation information on all instances of a particular program statement, or just a subset of instances, only one template activation operation is required, and only one template window is opened containing the requested information. The dynamic part of the template can be expanded, contracted and scrolled in order to bring the desired information into view, leaving out any unwanted instance information. Template expansion and contraction is initiated through a menu option in the template. Scrolling is controlled with scroll bars situated on the left side of each template's dynamic part.

Documentation template activation can take place in one of two ways: through the menus and by highlighting a statement keyword. The menu structure has been

defined to create a menu entry for every type of statement template and every type of program template. When a template is opened using its menu entry, the dynamic part of the template is sized so that all program instances of the template type are shown. Both program templates and statement templates can be activated using this method.

The other template activation approach involves highlighting the statement keyword of the statement in which documentation information is desired, and opening a menu and choosing the Activate Template menu item. The template is then opened up with the dynamic section resized to show documentation information pertaining only to the statement instance that was highlighted with the mouse. Because of the nature of this mode of template selection, only statement templates can be activated using this approach. Program templates do not always correspond directly with one program statement and thus cannot be activated in this way.

With an understanding of the documentation template structure now established, it is useful to define the distinguishing characteristics of statement templates and program templates. The template structure is identical for both the horizontal and vertical templates. However, the two template types are distinguished based on how the information is extracted from the program and how it is presented in the template. Each template type is discussed in the following sections.

6.2 A Framework For Studying Templates

Documentation templates can be described and compared by identifying the nature of processing necessary to determine values for each of the template's

fields. The processing required to obtain the value for a documentation template field can be described by identifying the information sources and establishing an interconnection of objects called processor elements that extract information from these sources and produce documentation values for their associated template field.

A processor element is an abstraction of the processing required to determine a template field entry, or the value of a statement field. It is supplied a set of inputs, and uses them to produce one or more outputs. By showing how these processors interconnect, one can gain an understanding of the complexity involved for determining documentation values. This form of representation is called a processing diagram. The building blocks for creating this kind of representation of template functionality is given in the following sections.

6.2.1 Information Sources

Information sources are discussed in detail in Section 5.1. There are three types: program statements, program topology, and the supporting system. The supporting system differs slightly from program statements and statement topology in the sense that information from the supporting system does not come from the program code. The information provided by this source is an integral part of each processor element. As a result, a processing diagram never has inputs or outputs labelled with supporting system as an information source. From the point of view of a processor element, the supporting system is an internal information source, while the program topology and statement fields are external information sources.

The sources used to determine the dynamic entries of a template field, are a good measure of the template's complexity and the nature of the documentation

provided. A template whose information sources are just fields of the statement being documented, would typically be at a lower level of complexity than a template whose field values are obtained by referencing the program topology. Furthermore, the nature of the documentation would generally be at a higher level when it involves an analysis of program topology.

6.2.2 Processor Types

There are two types of processor elements: statement field processors and template field processors. The task of the statement field processor is to determine the value of a Slam II statement field. If an explicit value for the field is given, the processor extracts this value. If an implicit value is assigned to the field, the statement processor performs the additional calculations necessary to determine the implicit value. If this involves internally processed information, or externally processed information, the necessary statement fields are referenced accordingly. In a processing diagram, a statement field processor is denoted by the label P_{si}, where i is a positive integer. The index i is used to distinguish between multiple statement field processors in a single diagram.

The function of the template field processor, is to determine a documentation value for one or more E/Slam template fields. This is done by referencing the appropriate information sources, and analyzing them with respect to the resident knowledge of the Slam II supporting system. When a template field processor requires information from a program statement field, it never accesses the field directly, instead, a statement field processor is placed in between. A template field processor is denoted by the label P_{ti} in a processing diagram, where i is

a positive integer. The index *i* is used to distinguish between multiple template field processors in a single diagram.

Both kinds of processors contain internal knowledge about the Slam II supporting system. The statement processors need this knowledge to be able to calculate values for defaulted statement fields. The template processors need the knowledge to be able to interpret statement field and topological information.

6.2.3 Processor Fan-In and Fan-Out

The fan-in of a processing element is a count of the number of inputs required for the processor to perform its function. It also provides rough measure of the degree, and the nature of processing required for a processor to produce its outputs.

The fan-in of a statement field processor, is a count of the number of external information sources that must be referenced, to determine the value of a statement field. Similarly, the fan-in of a template field processor identifies the number of external information sources that must be referenced, to determine the value of a template field.

The fan-out is a count of the number of outputs emanating from a processor. The fan-out of a statement field processor represents the value of a Slam II statement field. The fan-out of a template field processor is a count of the number of template fields that receive values as a direct result of a processor's computation. As seen in **Figure 6.2**, the processor element can be represented as a box with incoming arcs representing fan-in, and outgoing arcs representing fan-out.

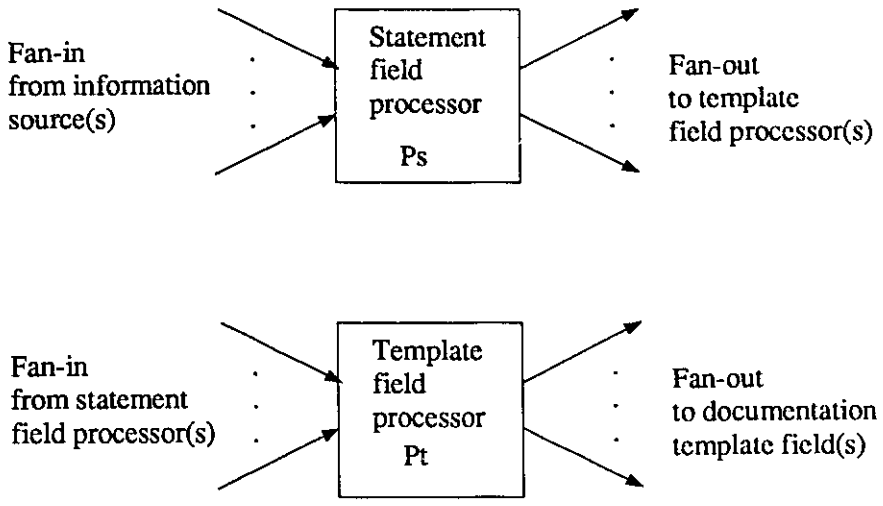


Figure 6.2 Fan-in and Fan-out of a statement field processor element and template field processor

6.2.4 Processor Operators

An additional level of understanding can be associated with processor elements by noting how the inputs and outputs of a processor are used. This is done by associating logical operators with processor input arcs and output arcs. The operators are expressed using the AND/OR formalism as a basis (Tanimoto 1987).

As seen in **Figure 6.4**, the logical operation is placed in a triangular node with inputs represented as incoming directed arcs, and outputs represented as outgoing directed arcs. The triangular node is attached to the processor element to show that the fan-in of the processor is the fan-in of the operator node. A similar configuration holds for the fan-out of a processor.

Operators associated with inputs to statement field processors, indicate how information is combined to determine a statement field value. Operators associated with template field processor inputs indicate how information is combined to

produce a documentation value. Operators associated with template field processor outputs indicate how the output information is distributed to a template field. The fan-out of a statement field processor is always one. Consequently, the output of a statement field processor never has an operator associated with it.

The logical operators are shown in **Figure 6.3** and are interpreted as follows. If the processor inputs must always be taken in combination, to determine an output value, then the element is drawn with the 'AND' operator positioned at the base of all mandatory inputs as shown in **Figure 6.4 a)**.

Logical Operation	Operator
AND	&
OR	+
XOR	%
NOT XOR	~%

Figure 6.3 Logical operators for expressing relationships between processor inputs and outputs

If only certain cases require the processor element to use all inputs, and in other cases, it only needs a subset of the inputs to determine a documentation value, the non-mandatory input is separated from the other inputs by the 'OR' operator as shown in **Figure 6.4 b)**. This kind of relation shows itself when a statement field processor element is required to determine the value of a statement field that can take an implicit specification. When the statement field contains an explicit value, the input that represents that value is immediately taken by the

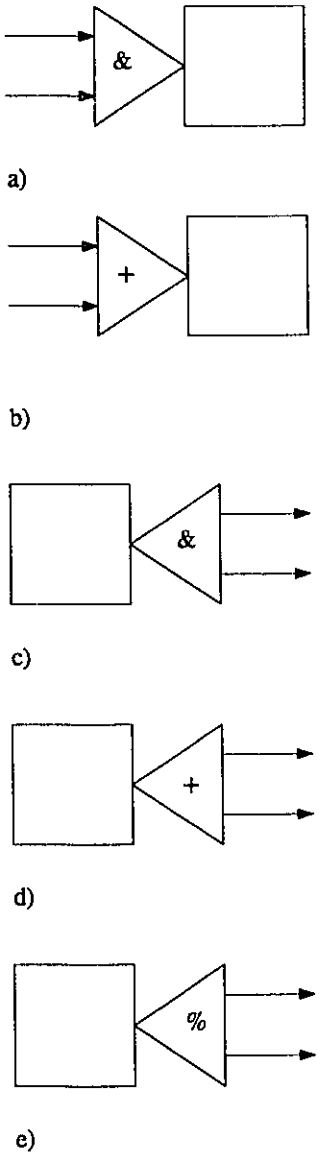


Figure 6.4 Processor element descriptions enhanced by using input/output operators. a) AND input operator, b) OR input operator, c) AND output operator, d) OR output operator, e) XOR output operator.

processor element. When the field takes on an implicit value, the processor element may require input information from another source in order to obtain a field value.

When the 'AND' operator is placed between the outputs of a processor element as seen in **Figure 6.4 c)**, it indicates that a value is produced for both outputs. This does not mean that both outputs are assigned the same value. Output leads of template field processors are never assigned the same value simultaneously, since this is an indication of redundant information. Instead, this kind of connection is used when it is necessary to split a template field in order to reduce complexity resulting from a processor element that has an abundance of information associated with it.

The 'OR' operator positioned between two output leads, as shown in **Figure 6.4 d)**, indicates a value is associated with either output or both outputs simultaneously. This indicates the associated documentation template fields could both be left blank, could both contain values simultaneously, or one field could contain a value while the other is left blank.

The 'XOR' operator positioned between two output leads, as shown in **Figure 6.4 e)**, indicates that a value is associated with at most one of the two outputs. This means the associated documentation template fields will never contain values simultaneously. This type of output is also used to reduce complexity by spanning information across template fields.

In the next section, the features of statement templates are discussed with the help of processing diagrams.

6.3 Information in Statement Templates and Relations Between Source and Target Information

A statement template is a medium for presenting the results of a program understanding effort, where all information contained within a given template, relates to a specific Slam II language statement. In comparison to program templates, statement templates can be viewed as a bottom-up approach to program understanding, since information is presented on a statement-by-statement basis where one builds an understanding of the entire program from information provided by each statement. Statement templates are also referred to as horizontal templates because of this statement-by-statement form of exposition.

A mapping exists between each field of a statement template and the information source used to obtain the value for the field. As seen in **Figure 6.5**, constructing a statement template involves extracting relevant information from the appropriate information sources, processing this information, and then entering it in the appropriate field of the template. The symbol **P** denotes a processor element which when expanded, reveals the template processor and field processor(s) used to calculate a template field value.

The three information sources are Slam II statement fields, Slam II program topology and the supporting system. If the value for a statement template field is obtained from a Slam II statement field, then it belongs to one of two categories: a field within the statement being documented (the focus statement), or a field outside of the statement being documented. A distinction is made between the two because the latter demonstrates an important feature of E/Slam - that a template

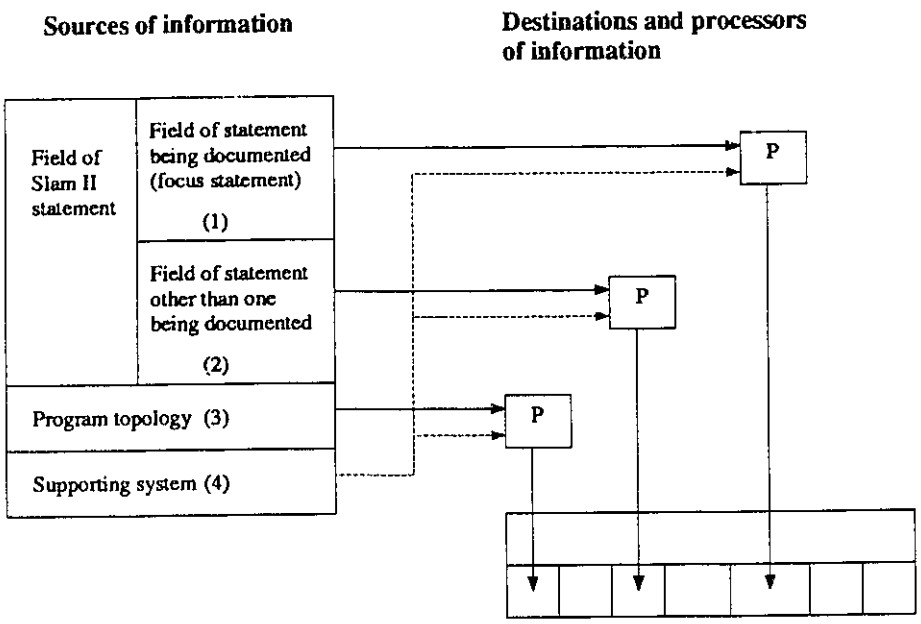


Figure 6.5 Information sources for E/Slam statement templates

can bring together information from statements elsewhere in the program in order to accompany the documentation of the focus statement.

Documentation templates can be compared by studying the information sources used to determine documentation values for their fields. They can also be compared by studying the processing required to produce documentation values, and the methods used for elucidating information. These three issues can be addressed by studying and comparing the template field processor elements associated with the documentation templates.

As shown in **Figure 6.6**, there are three primary configurations for the template field processor: single-input-single output, multiple-input-single-output, and multiple-output. A different kind of processing is associated with each configuration.

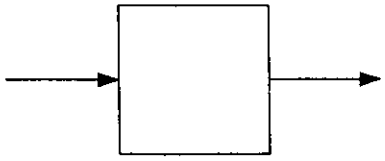
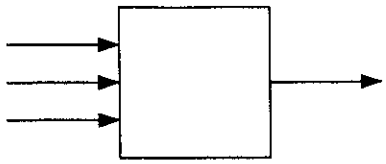
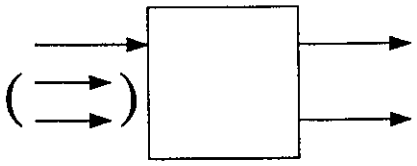
Sources of information	Type of statement template processor	Destination of Information
A field of the focus statement	 (single-input single-output processor)	A field of the correspond-ing statement template
A field of the focus statement + other sources (eg. other statement, program topology)	 (multi-input single-output processor)	A field of the statement template
A field of the focus statement + (optionally other sources) (eg. other statement, program topology)	 (multi-output and single or multi input)	Several fields of the statement template

Figure 6.6 Relationship between sources and destinations of information and types of statement template processors

For statement templates, a one-to-one correspondence exists between a statement field and a template field, when the only external source of information required by the template processor is a single statement field within the focus statement, and the only output of the processor is a single field within the focus statement's associated template.

A multiple-input-single-output relation exists when there are two or more external information sources used by a template field processor. In this configuration, one external information source is always a field of the focus statement. The other external source can be one or more fields of another program statement, the program topology, or a combination of the two. The output of the template processor is a field of the statement template for the focus statement.

A multiple-output relation exists when a single template processor simultaneously produces output for several fields of the statement template. In this category, the fan-in of the processor can be greater than or equal to one. The mandatory input is a field of the focus statement. In some situations additional input is required from one or more fields of another program statement, the program topology, or a combination of the two.

In Section 6.3.1 through 6.3.3 these three processor configurations are used to demonstrate the different kinds of information and information processing associated with E/Slam's statement templates.

6.3.1 Single-input-single-output Processor

The most commonly seen relation in statement templates, is a one-to-one correspondence between a field of the focus statement, and a template field. All fields in the CREATE template shown in **Figure 6.7** belong to this category.

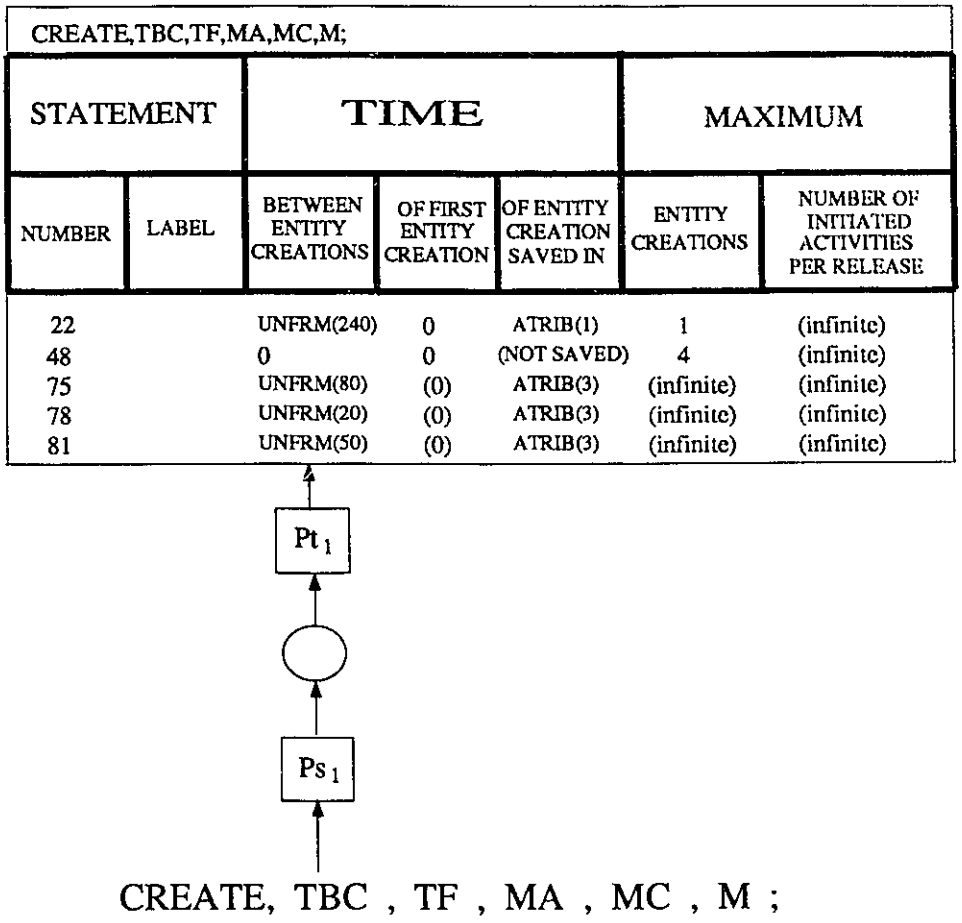


Figure 6.7 An example of a 1–to-1 correspondence between the TBC field of the CREATE statement and the "Time Between Entity Creations" field of the CREATE template

The processor relation for the "Time Between Entity Creations" field of the CREATE template shows that the only input required for the template field processor element to obtain the documentation value for the template field, is the value associated with the TBC field of the focus statement.

The responsibility of the statement field processor (P_{s1}) is to determine the value of the TBC field, and to make it available to the template field processor. Processor P_{s1} has built-in knowledge about the Slam II CREATE statement (i.e., knowledge from the supporting system). This gives the processor the ability to determine the correct value for the TBC field when the field has been implicitly specified. If the value in the TBC field is explicitly specified, the statement field processor P_{s1} extracts the value from the field and passes it on to the connected template field processor P_{t1}. If the TBC field is implicitly specified, the field value defaults to the constant value "infinity" . Knowledge about the default value is built into P_{s1}. Thus when P_{s1} senses a default in the TBC field it immediately outputs the value of infinity to P_{t1}. Processor P_{t1}'s function is to perform any additional processing necessary to get the input ready for insertion into the CREATE template. When the TBC field contains an explicit value, P_{t1} simply passes it unchanged to the output side of the processor. When TBC is implicitly specified, P_{t1} encloses its input in parenthesis giving the value "(infinite)", and this is passed as output.

The circular node placed between P_{t1} and P_{s1} represents the internal model of the Slam II program. The reason for having this is to show that the information required by the template processor is not obtained directly from the source statement. It is obtained from the internal model of the source statement. Drawing the relations in this way makes the processing diagram formalism more representative of the real system architecture.

6.3.2 Multiple-input-single-output Processor

There are three ways in which multiple-input-single-output processor configurations can be used to document a statement. The first involves enhancing documentation of the focus statement by including additional template fields that elucidate information about statements closely related to the focus statement. The multi-input-single-output processor is needed in this case to determine which statement is related to the focus statement.

The second use for a multi-input-single-output processor involves combining information from a field in the focus statement with information obtained from other statement fields in order to make the documentation value of the focus field more understandable. This could be done either by using the additional information to express the documentation value in more understandable terms. It could also be done by supplementing the documentation value with extra template fields that contain the additional information.

The third use for a multiple-input-single-output processor involves tapping into information supplied by the program topology. By using program topology as an information source, one is able to enhance the level of documentation supplied by a statement template. An example of each form of processing is provided in the following three sections.

Including Fields Of A Related Statement

When two statements are closely related, and information contained in one statement is useful in understanding the focus statement, it is advantageous to

include this information in the documentation template of the focus statement. This kind of interrelationship exists between the RECORD statement and the VAR statement as shown in **Figure 6.8.** These two statements are used to specify the collection of simulation run data for plotting. The RECORD statement defines the independent variable for the plot, while the VAR statement defines the dependent variable. Because these two statements are so closely coupled, the RECORD template contains a column entry for displaying the dependent variable for the plot and the VAR template contains a column for displaying the independent variable for the plot.

The supporting system states that the independent variable associated with a VAR statement is defined in the nearest RECORD statement that preceeds the VAR statement in the program. This knowledge is built into P_{t1}. What is needed to obtain the independent variable is the location of the focus statement in the program, and the program topology. The location of the focus statement is required because P_{t1} needs to know the position of the VAR statement before it can identify the associated RECORD statement. Once the position is found, the program topology is used to located the associated RECORD statement. Once it is located, P_{t1} accesses the value of the INDVAR field of the statement and displays it in the template column titled "Independent Variable For Output: Variable Name".

Combining Information To Supplement A Field

Situations arise where the quality of documentation on a particular statement field can be greatly enhanced by supplying additional information about the value present in the field. This additional information, obtained from elsewhere in the

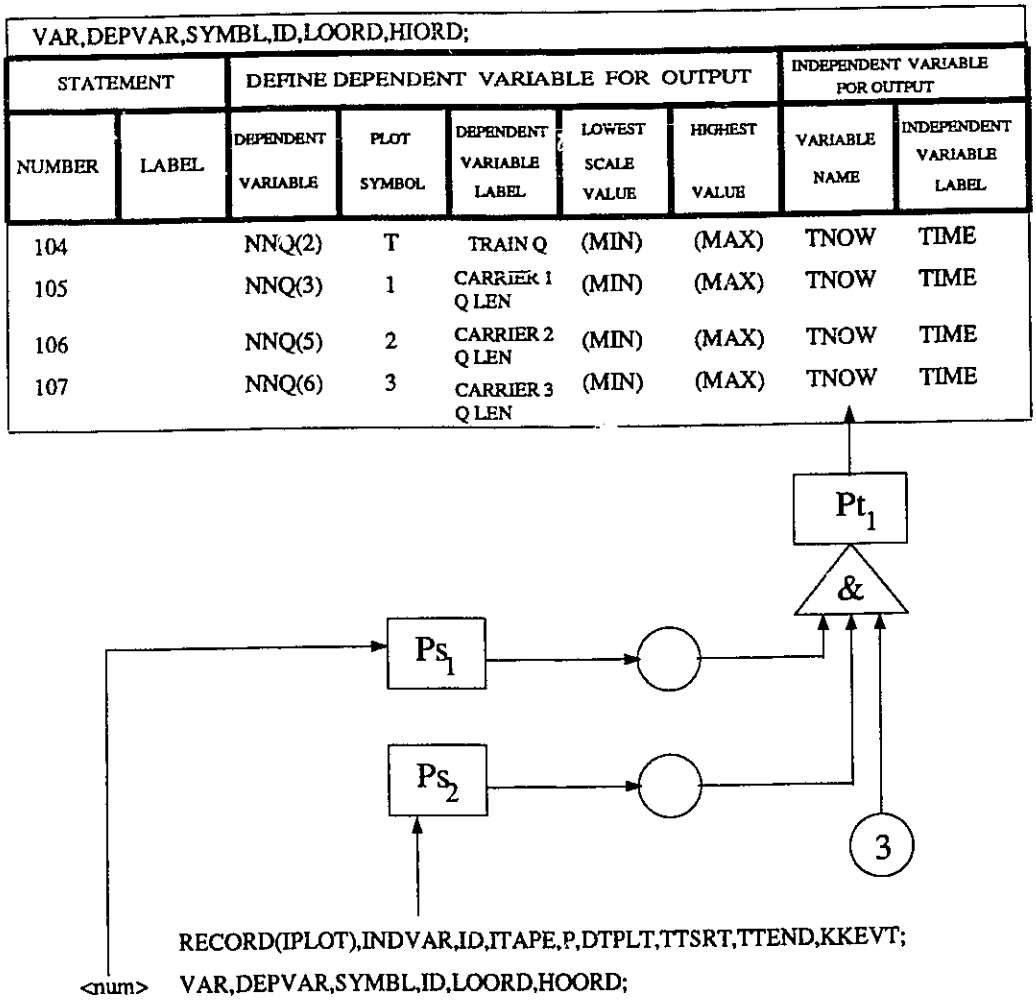


Figure 6.8 Sample VAR template, elucidating information from the Slam II program in **Figure 2.7**

program, could be positioned in the template field immediately beside the field with which we want to associate the supplemental information.

This situation occurs in the AWAIT template shown in **Figure 6.9**. Any file number listed in the IFL field of an AWAIT statement is associated with

either a resource or a gate. This is useful information for understanding the purpose of an AWAIT statement in the program, because it associates a meaning with the file number. For this reason, an additional column is positioned beside the template column that holds the file number. In this column, is placed the name of the block associated with the file number. The IFL field of the AWAIT statement maps onto the template field titled "Waiting-Queue File: Range: File Number". The supplemental information is placed beside it in the template field titled "Waiting-Queue File: Range: Name".

Processor P_{s3} extracts the file number associated with the IFL field of the AWAIT statement and makes it available to P_{t1}. P_{t1} uses this value as the search argument in the RESOURCE and GATE blocks. If we assume the value returned by P_{s3} is x, the IFL fields of all blocks are scanned for an occurrence of file number x. When a match occurs (represented by P_{s2}), the value in the RES or GAT field of the block is extracted and placed in the template field. The AND operator shows that all inputs are required to obtain a value for the template field. The program topology is not required to calculate the value of this field because, information on the relative ordering and placement of statements is not needed. The RESOURCE and GATE statements are global to the entire program. Thus it is just a matter of referencing the internal model of these statements directly.

It should be noted that the "Range: File Number" and "Range Name" fields are only filled in when the IFL field of the AWAIT statement contains an entry of the form "ATRIB(I)=J,K". When this occurs, file numbers in the range J to K are listed in the "Range: File Number" column. Immediately beside each file number is the name of the resource or gate with which it is associated.

AWAIT(IFL/QC),RESGATE/UR,BLOCKBALK,M;

STATEMENT		WAITING - QUEUE FILE				AWAIT ON		NUMBER OF RESOURCES REQUESTED	ACTION TAKEN ON FULL QUEUE	MAXIMUM NUMBER OF INITIATED ACTIVITIES PER RELEASE
NUMBER	LABEL	NUMBER	RANGE: FILE NUMBER	RANGE: NAME	CAPACITY	GATE	RESOURCE			
25		2			(infinite)		ATRIB(5)	(1)	(ENTITY DESTROYED)	(infinite)
51		3			(infinite)	POSITION		(1)	(ENTITY DESTROYED)	(infinite)
52		1			(infinite)		TNFORK		BALK(IDLE)	(infinite)
85		4			(infinite)		ATR(5)		(ENTITY DESTROYED)	(infinite)

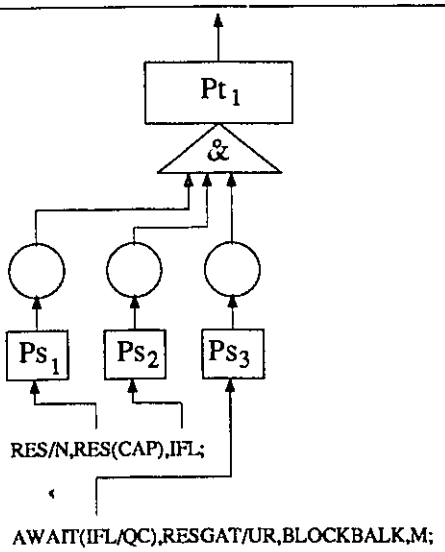


Figure 6.9 Sample AWAIT template, elucidating information from the Slam II program in **Figure 2.7**

Extra Information From Program Topology

The situation can arise where a Slam II statement has important implicit information related to program topology associated with it. Because the information is useful for enhancing understanding, it becomes necessary to provide a template column for displaying the information. An example of this is the "Activity: Type" column of the ACTIVITY template shown in **Figure 6.10**. This entry does not correspond to a specific parameter field within the ACTIVITY statement. Instead, its value is determined by the context in which the ACTIVITY statement is used.

In Slam II there are two kinds of activities: service activities and regular activities. A service activity is one which emanates from a QUEUE or SELECT node in the network model. Any other usage is classified as a regular activity. E/Slam determines the mode of usage and documents it in the "Type" column of the template with the text string "Regular" or "Service" depending upon its usage.

In order for the template processor to determine a value for this template field, it needs information about the focus statement, namely, its position in the program. This is obtained from processor P_{s1}. Secondly, it needs information about the program topology. P_{t1} uses the topological information to determine which network statement feeds into the focus statement. Once this is determined, the activity type is known and can be inserted into the template field. The AND operator shows that both information sources are necessary to determine a value for the field.

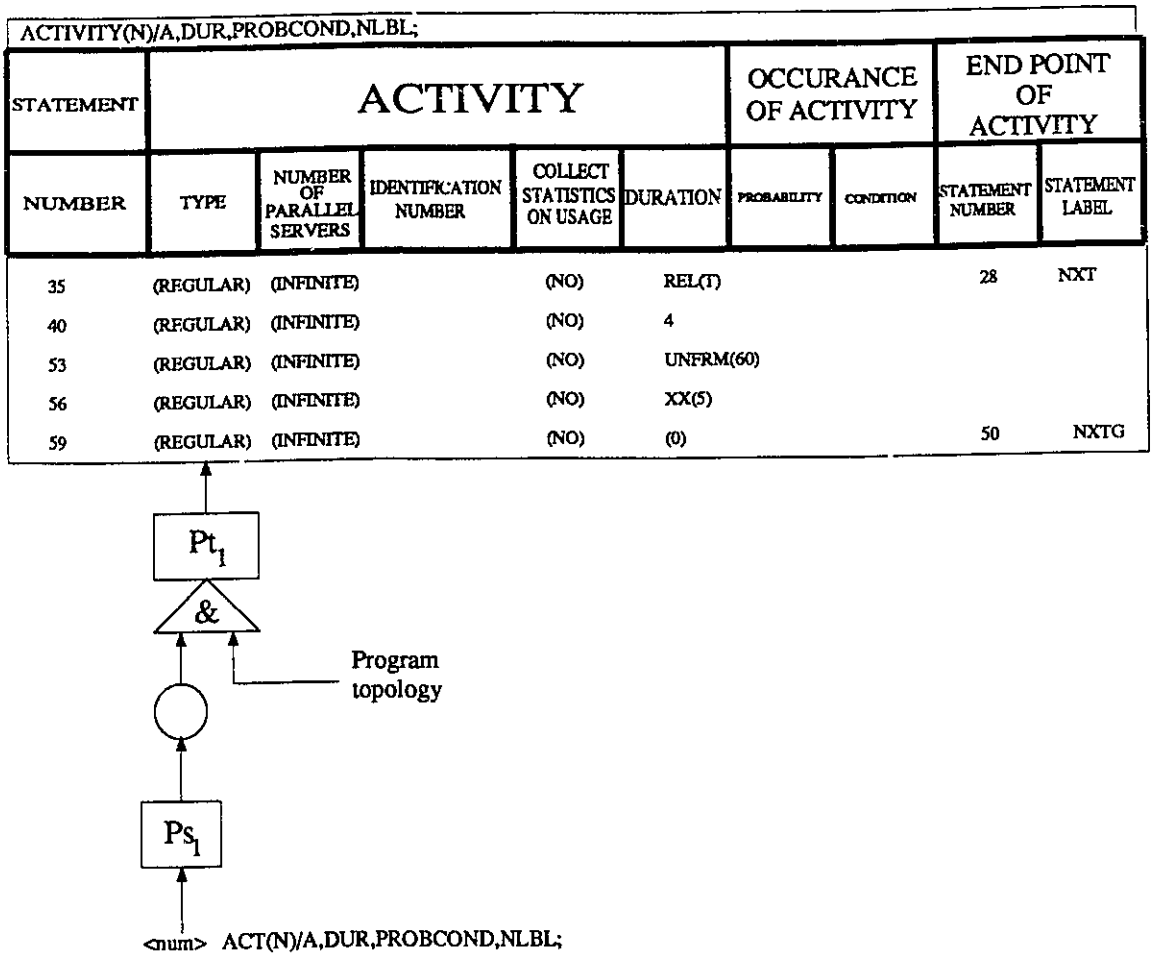


Figure 6.10 Sample ACTIVITY template and processing diagram for the Slam II program in **Figure 2.7**

6.3.3 Multiple-output Processor

On a few occasions, it is necessary to split information contained in a statement field and distribute it across multiple fields of the statement's documentation

template. This is required when the nature of the statement syntax makes understandability difficult. Field splitting is used when a statement field is overloaded, is a high volume field, or it contains cryptic information.

Overloaded Statement Field

An overloaded statement field is one that can take on more than one meaning. The meaning associated with the field at a particular time is a function of its content or the context in which it is used. In this situation, it is advantageous to provide additional template fields, one for each mode of use, and to enter the overloaded field's content into the appropriate template field. Doing this improves the documentation of the field because it makes the mode of usage explicit to the reader.

The RESGATE field of the AWAIT statement is an example of an overloaded field. The value specified in this field can either be the name of a Slam II gate, or a Slam II resource. To make the statement field more understandable, two template columns exist called "Await On: Gate" and "Await On: Resource". E/Slam determines the context in which the statement is being used and then displays the content of the RESGATE field in the appropriate template column. The AWAIT template is shown in **Figure 6.11**.

The method used to determine the context, depends upon the content of the RESGATE field. If the field contains a label name, it is necessary to look up the label name on the GATE and RESOURCE blocks in the program (P_{s1}). The

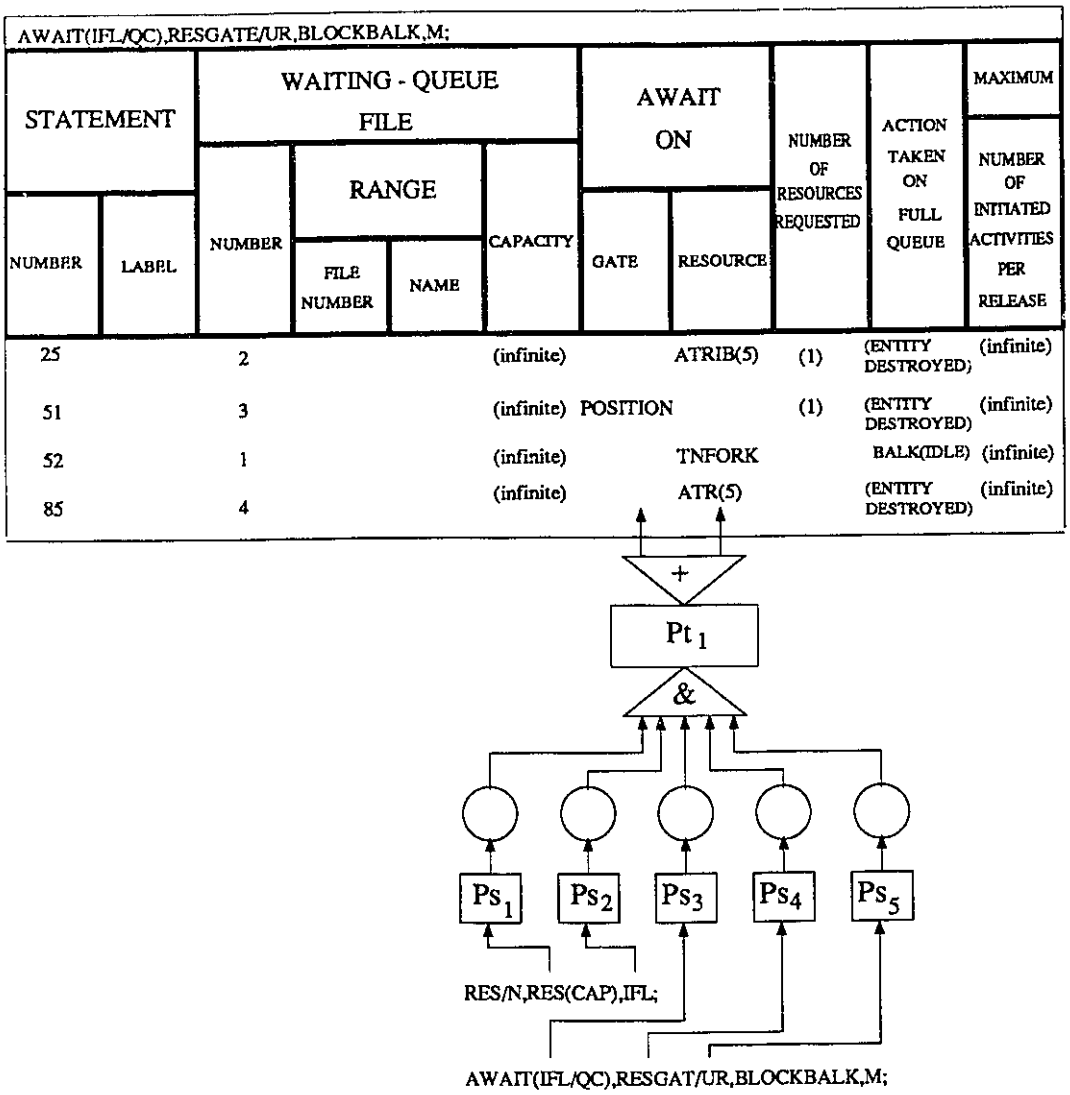


Figure 6.11 Sample AWAIT template and processing diagram for the Slam II program in **Figure 2.7**

statement in which the label is found dictates the mode in which the field is being used (i.e., whether the field refers to a gate or a resource).

If the RESGATE field contains a reference to the ATRIB() vector, P_{t1} must use the IFL field of the AWAIT statement (P_{s3}). If the IFL field contains an integer constant, then the mode in which the RESGATE field is being used is dictated by the block type in which this file number occurs (P_{s2}). If the file number is listed in a GATE block then the RESGATE field refers to a gate, if it is listed in a RESOURCE block then the RESGATE field refers to a resource.

If the RESGATE field of the AWAIT statement contains a reference to ATRIB(), and the IFL field is defaulted, then the supporting system dictates that the content of the RESGATE field refers to a resource.

If the RESGATE field contains a reference to the ATRIB() vector and the IFL field contains a range of file numbers, then all numbers listed in the range must be analyzed to determine what kind of blocks they are associated with. If it is a mix of gate blocks and resource blocks, then it may be that the RESGATE field is used in both modes during the course of a the simulation run, although it can only be in one mode at any given time. If it appears that both modes are used, then an additional test is made to determine if the RESGATE field applies to a resource only. This additional test involves checking whether or not the UR field of the AWAIT statement contains an explicit value (P_{s5}). If it does, then one can conclude that the RES/GAT field of the AWAIT statement is only used in resource mode. If the UR field contains an implicit value then one can only conclude that the RESGATE field is used in both modes.

The processing details just discussed are carried out by the template field processor P_{t1}. The Figure shows that if P_{t1} is supplied the value of the RESGATE field, the UR field, and the IFL field of the AWAIT statement, and the content of the RES field of RESOURCE blocks, content of the , GAT field of GATE blocks, and the IFL field of RESOURCE and GATE blocks, then it can determine a documentation value for the "Await On: Gate" and "Await On: Resource" fields of the AWAIT template. The OR operator is used to separate the two fields because it is possible to have an entry in both columns. When the analysis determines that the RESGATE field could be used in both modes, the content of the field is entered in both template columns.

High-volume Statement Field

A high volume field is one that contains a high degree of specification. In this situation, all information in the field is related to the field. However, there are components of information that are worth identifying individually. The IFL field of the QUEUE statement is an example of this. This field contains three components of information. When the IFL field contains the value "ATRIB(I)=J,K", it not only contains a file number specification "ATRIB(I)", it also contains a range specification "J" to "K", where J is the lower bound on "I" and "K" is the upper bound on "I". When this scenario arises, it is useful to separate this information and distribute it across the appropriate number of template fields. By doing this, one enhances understandability.

The QUEUE template shown in **Figure 6.12** uses this method. The "Waiting-

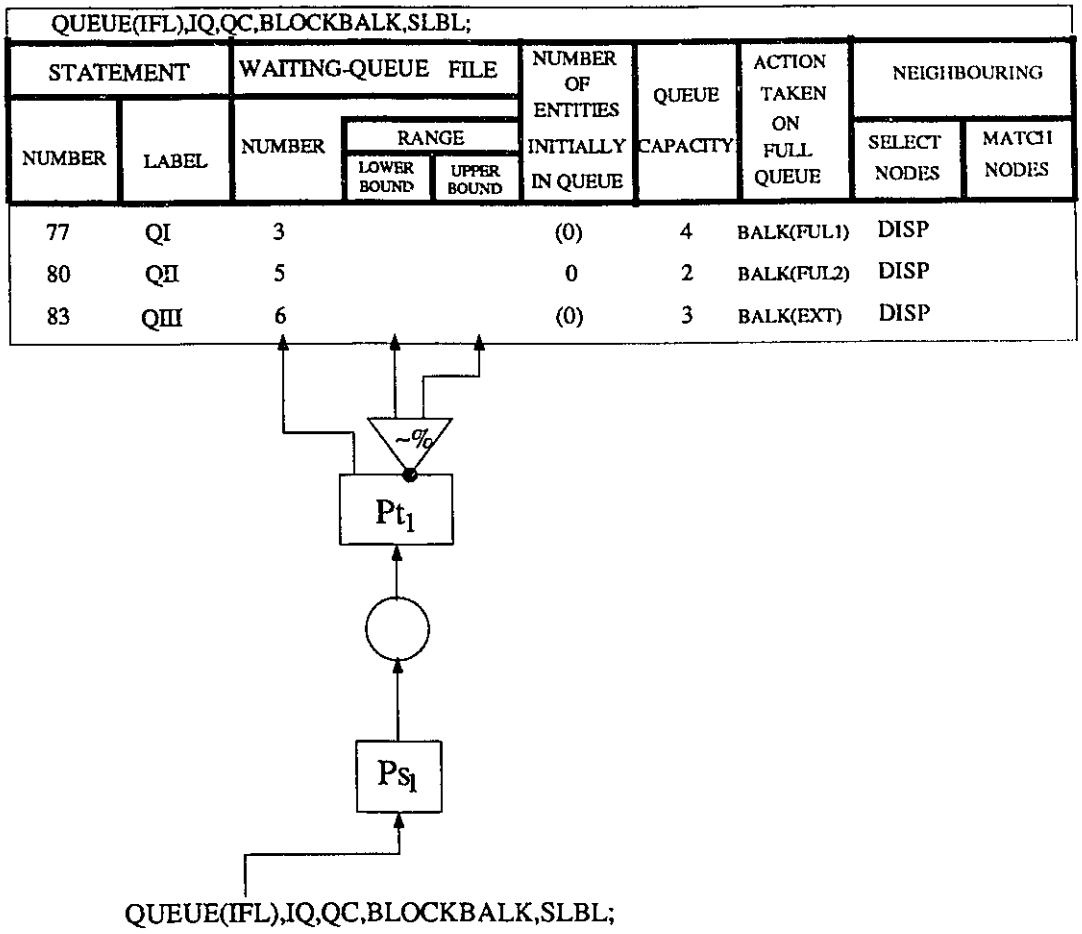


Figure 6.12 Sample QUEUE template and processing diagram for the Slam II program in **Figure 2.7**

Queue File" group contains three columns: "Number", "Range: Lower Bound", and "Range: Upper Bound". All three fields display information extracted from the queue file number field (IFL) of the QUEUE statement. The value of the IFL field in the focus statement is all that is required to determine values for the three outputs of P_{t1}.

The not-exclusive-or operator separating the "Lower Bound" and "Upper Bound" field outputs of P_{t1}, indicates either both template fields are assigned a value or neither are assigned a value. Slam II does not allow one bound to be specified without the other in the IFL statement field. No operator separates the "Number" output from the other two because P_{t1} always assigns a value to this field.

Cryptic Statement Field

The third situation that requires the field splitting technique to be used, arises when the information contained in a statement field is cryptic in nature. Some Slam II fields take on mnemonic values that can be difficult to understand. In order to make the documentation more user friendly in these instances, additional steps must be taken by the template processor.

There are two ways of handling the problem of cryptic fields. One method is to substitute a descriptive text phrase for the cryptic information and place it in the dynamic area of the template. The other method is to provide additional template columns for the cryptic field, and place additional descriptive information in the static part of the template.

In general, additional template columns are chosen over text phrase substitution, if the required phrase is too lengthy to place in the dynamic area of the template. However, there are other issues that must be taken into consideration before choosing one technique over the other. For instance, one would not adopt the field splitting technique if by inserting additional fields, the template size

becomes so large that it would take up too much screen space when displayed. Secondly, the nature of some fields may not allow one to span its content across only a small number of template fields. If a statement field can take on many different kinds of cryptic entries, then it is not feasible to supply a template column for each possibility.

The preemption rule (PR) field of the PREEMPT statement in **Figure 6.13** is an example where field splitting is used for cryptic reasons. The PREEMPT statement is used to preempt resources from entities. The PR field allows one to specify a preemption rule of "HIGH(I)" or "LOW(I)". "HIGH(I)" indicates priority is given to the entity with the higher value of ATRIB(I). Similarly, "LOW(I)" indicates that priority is given to the entity with the lower value of ATRIB(I). To make this more explicit, two fields are present in the PREEMPT template. If "LOW(I)" is present in the PR field then the value "ATRIB(I)" is entered in the template field titled "Priority Among Preempt Entities Given To Entity With: Lowest Value Of". If the value is "HIGH(I)", then the value ATRIB(I)" is entered in the field titled "Priority Among Preempt Entities Given To Entity With: Highest Value Of".

To determine the value for these template fields, the only input required by P_{t1} is the value of the PR field in the focus statement. Once this value is provided P_{t1} supplies values for its two outputs. When an explicit specification is present in the PR field, P_{t1} assigns a documentation value to only one of its outputs, the other is left blank. If the PR field is defaulted (implicit specification), then P_{t1} produces the output "(NO PRIORITY)" for both outputs, since the supporting system dictates that this is the default for an unspecified PR field.

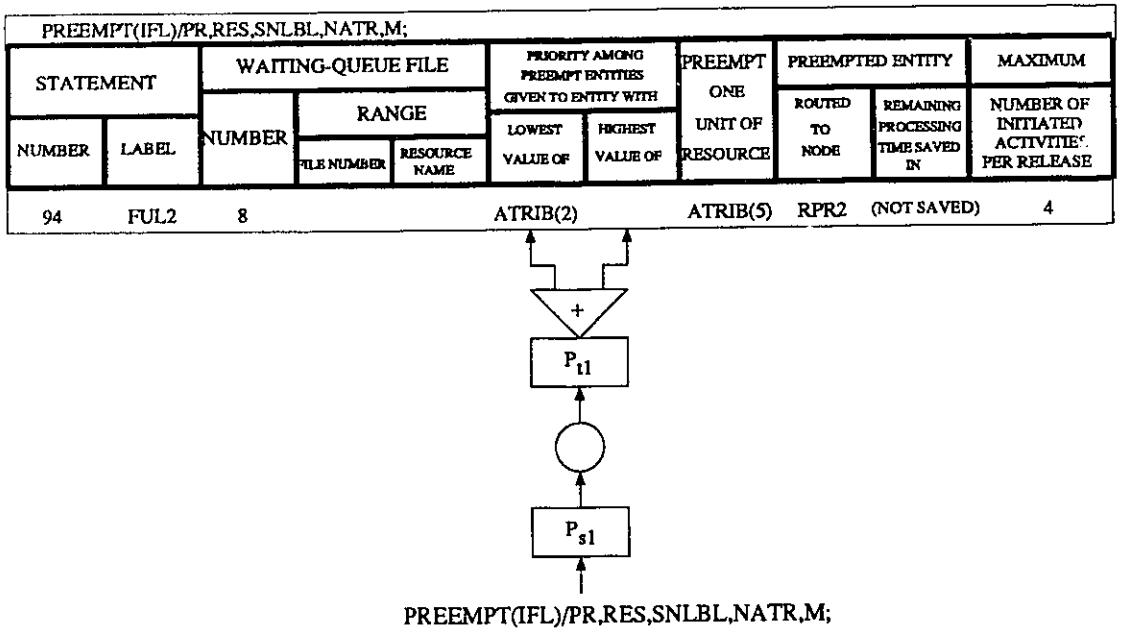


Figure 6.13 Sample PREEMPT template and processing diagram for the Slam II program in **Figure 2.7**

6.4 Information in Program Templates

The following section discusses the concept of program templates. Program templates elucidate simulation program information from a point of reference orthogonal to that used for generating statement templates. There are two goals to this section. First, to show how elucidating program information from multiple perspectives enhances understandability. Second, to demonstrate a perspective that is ideal for elucidating information in conventional simulation programs. In order to make the concepts clearer, examples of program templates are provided. The templates are filled in with information extracted from the Slam II program in Chapter 2.

6.4.1 Points of Reference and Program Understanding

In the field of mechanical engineering, when the engineer designs a physical object, in order for the engineer to impose a feeling of complete understanding of the object to others, he finds it necessary to create drawings of the object from different view points. He realizes that this is necessary since any one point of view does not give a complete description of the object. Consequently, a typical engineering drawing of any moderately complex object consists of a presentation from more than one projection plane. Furthermore, the engineer knows that it is best to choose points of reference that are orthogonal to each other, because it minimizes duplication of information supplied by other projection planes. By taking this approach when describing an object to others, the engineer imposes a more complete, more concise and a clearer understanding in the mind of the learner than would have been achieved by supplying only one projection of the object.

This principle for describing objects also applies to documenting simulation programs. More than one viewpoint is necessary to gain a thorough understanding of a simulation program. Documenting from only one point of reference will cause the learner to miss valuable information since one projection plane typically can not elucidate all information about the program. Furthermore, it seems that orthogonal projections facilitate the presentation of information in a clearer, more concise and more complete way because it minimizes duplication.

Depending upon the perspective chosen for observing an entity, different a priori knowledge is often necessary on behalf of the observer in order to fully understand what is being observed. In the case of a software system for example,

one has the option to go directly to the implementation level, read the program code and build a conceptual understanding from this. Another approach is to go to the architectural design documents and avoid the program code entirely. Each is a valid approach to learning. However, there is one key distinguishing factor: the a priori knowledge required on behalf of the learner is not the same. One approach requires the reader to know the implementation language and the other does not.

The previous example introduces another issue relating to points of reference (i.e., abstraction levels). Different levels of abstraction can be chosen when describing an object. If an engineer is describing a television set, one of his drawings may describe an external frontal view of the object showing the control panel, switches and knobs. This is fine for the average user of the television set. No further information is required to understand and use the television. The television repairman however needs an understanding at a more detailed level of abstraction. Consequently, he is supplied with schematic wiring diagrams that enable him to understand the inner workings of the television. To the average user, the wiring schematics would be an overload of unnecessary information. To the repairman the external views of the television offer insufficient information. The engineer's solution is to know who his audiences are, and to supply appropriate information at the right abstraction level to each audience.

This issue of abstraction levels also applies to the analyst who wishes to understand a simulation program. If the analyst simply wishes to exercise the simulation model under a new set of experiments, he should be able to specify the new parameters of experimentation without having to understand the implementation language. The programmer who wishes to modify the program

model or locate and fix a bug in the program needs more detailed information and thus is required to study the implementation details of the program, interacting directly with the program code.

Program understanding systems must be cognizant of their audience and elucidate information at an abstraction level that is appropriate for their audience. By having multiple abstraction levels available for viewing an entity, the appropriate one can be chosen according to the amount of detail the analyst desires about the subject entity. At a given abstraction level, the program understanding system should have the ability to elucidate information from different projection planes. Information would be observed at the same level of detail regardless of the perspective chosen, however the information would be organized, and presented in different ways.

The program understanding system handles the task of analyzing the program, organizing and clarifying the information contained within it, and presenting it to the user in a form that matches the observational point of reference chosen by the user. This allows the analyst to focus more on understanding and less on locating the relevant information required for understanding. Of course, to use this kind of approach the designers of the program understanding system must know what points of perspective are useful in observing the system.

In computerized simulation, a frame of reference ideal for making projections is from the perspective of a simulation program's functional elements. All conventional simulation programs can be studied using this frame of reference. Thus an analyst who is familiar with simulation, could learn a simulation program

using this frame of reference. The functional elements of conventional simulation programs are discussed below.

6.4.2 Functional Elements of Conventional Simulation Programs

Conventional simulation programs can be partitioned into the following functional elements: model structure, model output, input scheduling, initialization conditions, termination conditions, and data collection and display. The term experimental frame is used to represent input scheduling initialization conditions, termination conditions, data collection and display specifications for a simulation run. Each element is discussed in turn below, with specific references to Slam II.

Model Structure

An element of a simulation program is part of the model structure if it contributes to the specification of either the static or dynamic characteristics of the model. The component models, input, output and state variables associated with the component models comprise the static characteristics of the model. The dynamic characteristics constitute the rules of interaction between component models. The topology of the Slam II network model is part of the model structure. GATE definitions, RESOURCE definitions, and ACTIVITY specifications also contribute to model structure.

Model Output

An element of a simulation program belongs to the category of model output if it is in some way related to the model output variables or output functions in the program. This includes the specification of output variables and the procedures for their computation. The Slam II COLCT, RECORD, VAR, TIMST, STAT and ACTIVITY statements fall under this category.

Input Scheduling

An element of a simulation program is part of input scheduling if it has to do with the time related scheduling of external inputs to the system. The CREATE statement which schedules the arrival of entities, and the DETECT statement which injects entities into the network under specified conditions fall into this category.

Initialization

An element of a simulation program belongs to the category of initialization if it has to do with the setting of program elements to their initial values. This includes the setting of model variables to initial values and the initialization of storage files, and initialization of random number streams to name a few. Fields from the Slam II GATE and QUEUE statements fall into this category. Many of the Slam II control statements such as ARRAY, ENTRY, INIT , INTLC, and SEEDS statements also fall into this category.

Termination Condition

An element of a simulation program is a termination condition if it in some way contributes to the specification of the conditions under which either a simulation run or the simulation study will end. This includes the specification of run stop time, termination based on thresholds or termination based upon the occurrence of a specific event. The TERMINATE and INIT statements are contributors to this category.

Behavior Processing

An element of a simulation program falls under the category of behavior processing and display if it has to do with the specification for the collection, analysis and formatting of model output or the presentation of this output. The Slam II COLCT, RECORD, VAR, TIMST, STAT and ACTIVITY statements fall under this category.

6.4.3 Gleaning The Information

The program perspective involves viewing the simulation program from a perspective orthogonal to that of statement templates, yet at the same abstraction level. Instead of presenting program information on a statement-by-statement basis, it is expressed according to the functional elements of a simulation program.

In conventional simulation programs it is often the case that these primary functional elements are manifested in different parts of the program. This is a common occurrence in simulation languages like GPSS, SIMAN and Slam II.

Figure 6.14 shows how information from the various categories is distributed throughout the example program presented in Chapter 2. Each category is listed in a column, to the right of each statement. If the statement on a particular line contains a specification for one or more of the six functional elements, then an X is placed on that line under the appropriate column.

```
;*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*1   * Model structure.
;   MODEL OF LOADING DOCK OPERATION        .                         2   |   * Output variables and functions.
;   XX(1) :  NUMBER OF TRAIN CARS LEFT TO UNLOAD.                    3   |   |   * Input scheduling.
;   XX(2) :  SUBAREA ALTERATION AMOUNT.                              4   |   |   |   * Initialization.
;   XX(3) :  USED IN FORKLIFT NUMBER ASSIGNMENT.                     5   |   |   |   |   * Termination.
;   XX(4) :  NUMBER OF FORK LIFTS AVAILABLE FOR UNLOAD OPERATIONS.   6   |   |   |   |   |   * Scheduled data collection,
;   XX(5) :  DURATION OF TRAIN REPOSITIONING OPERATION.              7   |   |   |   |   |   | reduction and display.
;*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*8   |   |   |   |   |   |
GEN,WENDT,LOADING DOCK,02/11/92,2,Y,,Y,Y,N,72;                       9               X       X
LIMITS,8,5,100;                                                     10   X
EQUIV/XX(5),REPOS/XX(1),TRAINCARS;                                  11               X
NETWORK;                                                            12   X
      GATE/POSITION,CLOSE,3;                                        13   X           X
      RES/TNFORK(1),1;                                              14   X           X
      RES/DOCK,2;                                                   15   X           X
      RES/4,TKFORK(4),4;                                            16   X           X
      RES/5,TKFORK2(6),4;                                           17   X           X
; TRAIN POSITIONING OPERATIONS:                                     18
;        ATR(1)     :     TRAIN ARRIVAL TIME.                       19
;        ATR(2)     :     TRAIN LENGTH.                             20
;                                                                   21
      CREATE,UNFRM(240),0,1,1;                                      22   X       X
      ASSIGN,ATRIB(2) = EXPON(30);                                  23   X
      ASSIGN,TRAINCARS = ATR(2),ATR(5) = 2;                         24   X
      AWAIT(2),ATRIB(5);                                            25   X
      ASSIGN,XX(2) = XX(4) - NNRSC(TNFORK);                         26   X
      ALTER,TNFORK/XX(2);                                           27   X
NXT   GOON,1;                                                       28   X
      ACT,,TRAINCARS.LE.0,DPRT;                                     29   X                   X
      ACT,,TRAINCARS.GE.XX(4),RDY;                                  30   X                   X
      ACTIV,,TRAINCARS.GT.0.AND.TRAINCARS.LT.XX(4);                 31   X                   X
      ASSIGN,XX(2) = XX(1) - XX(4);                                 32   X
      ALTER,TNFORK,XX(2);                                           33   X
RDY   OPEN,POSITION;                                                34   X
      ACT,REL(T),,NXT;                                              35   X                   X
```

Figure 6.14 Example program from Chapter 2, showing how the six primary functional elements are manifested in different parts of the simulation model (Continued . . .)

```
;
; TRAIN PULLS AWAY:
;
DPRT   GOON;
       ACT,4;
       FREE,DOCK;
       COLCT(1),INT(1),TRAIN SVC TIME,8/15/15;
       TERM,20;
; TRAIN UNLOADING  OPERATIONS:
;        ATR(1)     :     FORKLIFT NUMBER.
;        ATR(2)     :     BATCHED ENTITY REFERENCE.
;
       CREATE,0,0,,4;
       ASSIGN,XX(3) = XX(3) + 1,ATR(1) = XX(3);
NXTG   GOON;
       AWAIT(3),POSITION;
       AWAIT(1),TNFORK,BALK(IDLE);
       ACT,UNFRM(60);                          UNLOAD OPERATIONS.
IDLE   BATCH,,4,,,ALL(2);
       CLOSE,POSITION;
       ACT,XX(5);                              TRAIN MOVES FORWARD.
       UNBATCH,2;
T      GOON;
       ACT,,,NXTG;
ENDNETWORK;
```

```
                * Model structure.
                |    * Output variables and functions.
                |    |    * Input scheduling.
                |    |    |    * Initialization.
                |    |    |    |    * Termination.
                |    |    |    |    |    * Scheduled data collection,
                |    |    |    |    |    | reduction and display.
36              |    |    |    |    |    |
37              |    |    |    |    |    |
38              |    |    |    |    |    |
39              X
40              X                              X
41              X
42              X    X                         X
43              X                   X
44
45
46
47
48              X         X
49              X
50              X
51              X
52              X
53              X                              X
54              X
55              X
56              X                              X
57              X
58              X
59              X                              X
60              X
```

Figure 6.14 Example program from Chapter 2, showing how the six primary functional elements are manifested in different parts of the simulation model (Continued . . .)

```
;
;   TRUCK LOADING OPERATIONS:
;   ATR(2) :  TRUCK CARRIER NUMBER.
;   ATR(3) :  TRUCK ARRIVAL TIME.
;   ATR(4) :  TIME REQUIRED TO UNLOAD TRUCK.
;   ATR(5) :  RESOURCE NUMBER TO SEIZE
;
;   FILE 1 :  QUE FOR CARRIER I
;        2 :   "          "    II
;        3 :   "          "    III
;        4 :  WAITING QUEUE FOR TKFORK
;        5 :  FULL QUEUE CARRIER I
;        6 :   "             "    II
;
      CREATE,UNFRM(80),,3;                    ARRIVALS OF CARRIER 1
      ASSIGN,ATR(2)=1,ATR(4)=UNFRM(40),ATR(5)=XX(4);
QI    QUE(3),,4,BALK(FUL1),DISP;
      CREATE,UNFRM(20),,3;                    ARRIVALS OF CARRIER 2
      ASSIGN,ATR(2)=2,ATR(4)=UNFRM(30),ATR(5)=XX(4);
QII   QUE(5),0,2,BALK(FUL2),DISP;
      CREATE,UNFRM(50),,3;                    ARRIVALS OF CARRIER 3
      ASSIGN,ATR(22)=3,ATR(4)=UNFRM(60),ATR(5)=XX(4);
QIII  QUE(6),,3,BALK(EXT),DISP;
DISP  SELECT,CYC,,QI,QII,QIII;
UNLD  AWA(4),ATR(5);
      ACT,ATR(4);
      FREE,ATR(5);
      COLCT,INT(2),TRUCK SVC TIME;
      TERM;
```

```
61
62
63
64
65
66
67
68   * Model structure.
69   |    * Output variables and functions.
70   |    |    * Input scheduling.
71   |    |    |    * Initialization.
72   |    |    |    |    * Termination.
73   |    |    |    |    |    * Scheduled data collection,
74   |    |    |    |    |    | reduction and display.
75   X         X
76   X
77   X         X
78   X         X
79   X
80   X         X
81   X         X
82   X
83   X         X
84   X
85   X
86   X                             X
87   X
88   X    X                        X
89   X                   X
```

Figure 6.14 Example program from Chapter 2, showing how the six primary functional elements are manifested in different parts of the simulation model (Continued . . .)

```
FUL1   PREEMPT(7)/LOW(2),ATR(5),RPR1,,4;
       ACT,,,UNLD;
RPR1   ASSI,ATR(2)=0;
       ACT,,,QI;
FUL2   PREEMPT(8)/LOW(2),ATR(5),RPR2,4;
       ACT,,,UNLD;
RPR2   ASSIGN,ATRIB(2)=0;
       ACT,,,QII;
EXT    TERM;
;
; XX(4) : RESOURCE NUMBER TO USE IN RUN
;
MONTR,TRACE;
REC,THOW,TIME,P;
VAR,NNQ(2),T,TRAIN QUEUE;
VAR,NNQ(3),1,CARRIER 1 Q LEN;
VAR,NNQ(5),2,CARRIER 2 Q LEN;
VAR,NNQ(6),3,CARRIER 3 Q LEN;
INIT,,5000,Y/1,,N;
INTLC,XX(3)= 0,XX(4)=4,XX(5)=5;
SIM;
INIT,,5000;
INTLC,XX(3)=0,XX(4)=5;
SIM;
INIT,100,5000;
INTLC,XX(4)=4;                  4 FORK LIFTS
SIMULATE;
INTLC,XX(4)=5;                  6 FORK LIFTS
SIM;
```

```
                * Model structure.
                |    * Output variables and functions.
                |    |    * Input scheduling.
                |    |    |    * Initialization.
                |    |    |    |    * Termination.
                |    |    |    |    |    * Scheduled data collection,
                |    |    |    |    |    | reduction and display.
 90    X
 91    X                                        X
 92    X
 93    X                                        X
 94    X
 95    X                                        X
 96    X
 97    X                                        X
 98    X                         X
 99
100
101
10:                                           X
10?                                           X
104                                           X
105                                           X
106                                           X
107                                           X
108                    X                      X
109                    X
110   X
111                    X                      X
112                    X
113   X
114                    X                      X
115                    X
116   X
117                    X
118   X
```

Figure 6.14 Example program from Chapter 2, showing how the six primary functional elements are manifested in different parts of the simulation model

In **Figure 6.14** one can visually see the two approaches to E/Slam's documentation. Statement templates correspond with information on a statement-by-statement, line-by-line (horizontal) basis in the Slam II program. Program templates present information on a column by column (vertical), or functional element perspective where one takes a vertical slice of the program. This is the essence of orthogonality between the two approaches to documentation.

6.4.4 E/Slam Program Template Menu Structure

E/Slam's program template menu hierarchy is designed using the functional elements of simulation programs as a basis. The upper levels identify general concepts and the lower ones identify concepts more refined, some of which are unique to the Slam II language. Model structure is not included in the menu hierarchy since just about every statement in a Slam II program contributes to model structure. If the analyst wishes to gain information on model structure he would use E/Slam to generate a network graph of the simulation model and/or go directly to the program source code displayed in one of E/Slam's windows. Statement templates could also be used to view this kind of information. To reduce menu complexity, termination conditions are included with initialization conditions under the menu item titled "Simulation Run". Similarly model output is include under the menu structure titled "Data Collection".

As shown in **Figure 6.15**, the program perspective involves partitioning information into two major categories: Experimental Frame and Usages. The Experimental Frame partition provides a pathway to simulation program information based upon the function elements of conventional simulation programs. The Usages partition provides a vertical slicing of the program based upon the points

of reference of various resource elements in the program. A resource element is a component of a programming language that serves as a holding place for information and can be drawn upon at different locations in the program. In Slam II, the following are resource elements: identifiers, files, resource blocks, gate blocks, and entities.

The simulation methodology becomes more refined as one follows through the hierarchy that descends from Experimental Frame. "An experimental frame defines a limited set of circumstances under which the system is to be observed or subjected to experimentation" (Ören, Zeigler 1979, p69–82). Issues that fall under Experimental Frame involve the initialization of program elements prior to simulation run and simulation study commencement. Scheduling of various forms of input for the simulation run, collection of data and generation of output information is also included.

Initialization is partitioned into initialization of experimentation parameters, initialization of identifiers, initialization of files, and initialization of gates.

Initialization of parameters of experimentation involves the initialization of functional elements that control the simulation. This includes the initialization of parameters that control the simulation run, and parameters that control the continuous component model for continuous simulation. Initialization of random number streams is included here.

Issues that fall under simulation run initialization include simulation start times and termination conditions. Parameters of experimentation that fall under the continuous model heading include the specification of minimum and maximum step sizes taken by the integration algorithm in computing the next states. Communi-

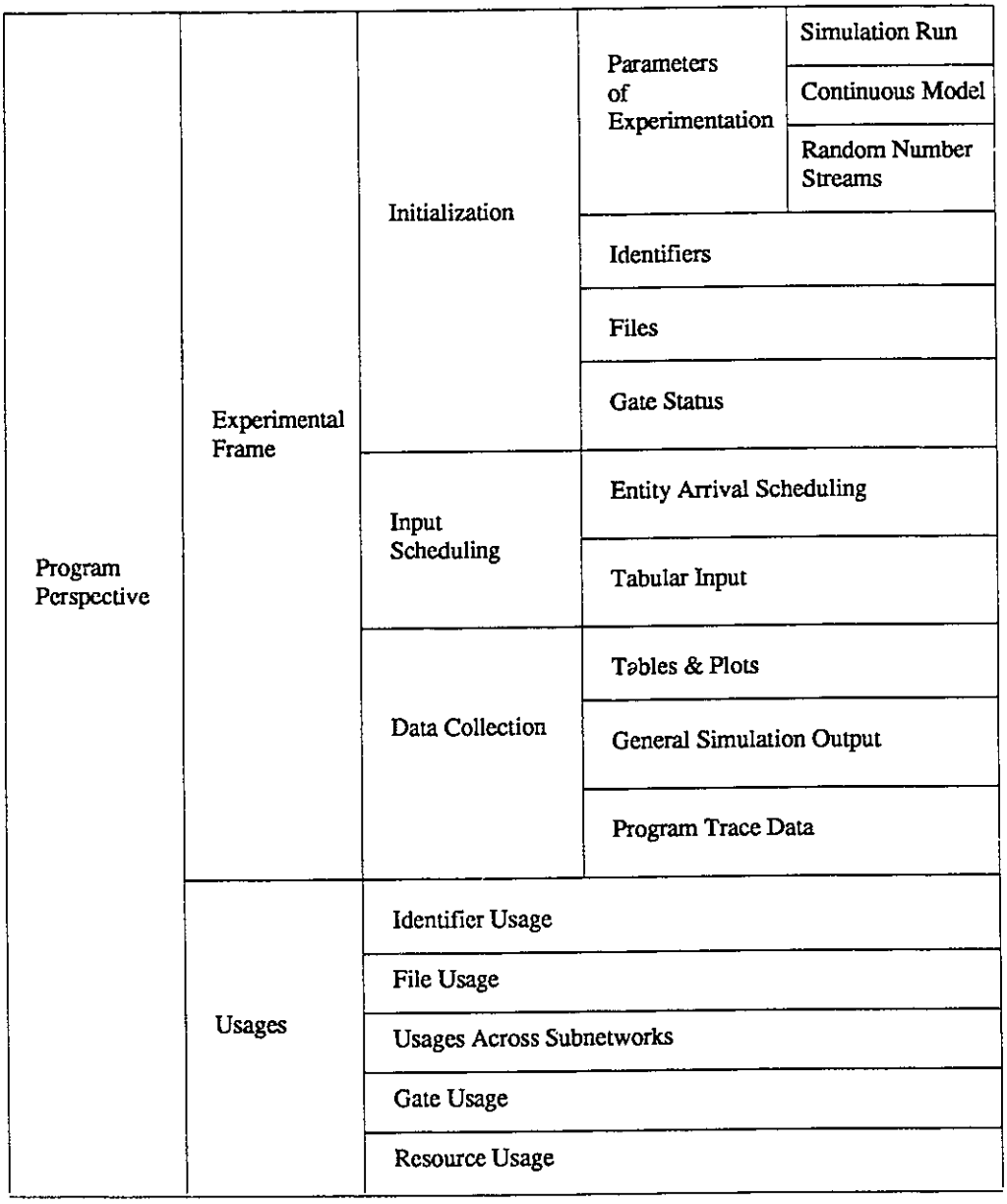


Figure 6.15 E/Slam program template menu structure

cation intervals for data value recording also fall under this category. Numerical integration accuracy specifications are defined here as well as the specification of the integration algorithm used for calculating the state trajectories.

Random number stream initialization involves specification of the seed value used for each stream, as well as an indication of whether or not the stream should be reinitialized between simulation runs.

Identifier initialization pertains to the assignment of initial values to identifiers used in the program. This includes the Slam II global variables II, XX, SS, DD, and ARRAY.

File initialization involves specifying the initial state of storage files for each run. This includes defining whether or not data collection files and entity queues are to be cleared between runs. This also includes specifications involving the insertion of entities directly into files prior to simulation run commencement.

Gate initialization involves stating the original disposition of a Slam II GATE as being open or closed.

Input scheduling involves specifying the time series of external inputs to the model. Input scheduling in Slam II is realized in two ways. One manifestation is through the scheduling of entity arrivals to the network model during the course of the simulation run. The second is through the use of data tables that provide the simulation model with external data.

The third partition under Experimental Frame is date collection. Under this heading one can view specifications for data collection to be carried out during the simulation run. This includes specification of tables and graphs used to displayed the collected data.

In the context of Slam II data collection can be partitioned into three components: tables and plots, general simulation output, and program trace data. The tables and plots category covers specifications for the collection of data to be used for decision making purposes.

The general simulation output category involves specifying whether reports about the simulation itself are to be generated. This includes the program statement listing, Slam II echo report and Slam II summary report. The echo report shows Slam II's interpretation of the simulation model. It basically summarizes the values specified in various Slam II statements by group. Possible groups include general options, continuous variables, state events, data recording, and number streams The Slam II summary report shows statistical results of the simulation. The report consists of a general section followed by the statistical results for the simulation categorized by type.

The program trace category lists specifications related to the collection of data for debugging purposes. This includes specifications for listing file contents, values of state variables, variable tracing, and node tracing.

The "Usage" portion of the menu structure provides information on the utilization of program resource elements. A program resource element is an atomic element of the programming language and is used as a holding place for data. Identifiers, files, resources, gates and entities are Slam II resource elements. A simulation program utilizes these elements routinely during its operation. References to them are often scattered throughout the simulation program, much like references to the functional elements of simulation programs. A significant amount of knowledge about a simulation program can be gained by

concentrating on a resource element and observing how and where it is used in the program. This is the motivation for offering "usage" templates.

The "Identifier Usage" category lists for each program identifier, the locations in the program where that identifier has been referenced. Similarly, the "File Usage" category lists for each file, the locations in the program where it is used.

The "Usages Across Subnetworks" views the network model as a series of disjoint connected graphs and identifies for each disjoint graph, the resource elements it shares in common with other disjoint graphs in the model.

The "Gate Usage" category lists, for each gate, all locations in the program where that gate is referenced. Similarly, the "Resource Usage" category lists, for each resource, all locations in the program where that resource is referenced.

Some examples of the different categories of templates with reference to the Slam II program in Chapter 2, are given in the following sections.

6.4.5 Initialization Templates

Run Control Initialization

The Run Control template like most program templates, obtains its information from a number of locations in the Slam II program. A Sample Run Control template is shown in **Figure 6.16.**

E/Slam determines the run number each run control specification applies to by observing the position of INIT statements relative to the position of SIMULATE statements in the program, taken in combination with the value of the NRUN field in the GEN statement. The processor element that generates values for the "Run

INITIALIZATION OF PARAMETERS OF EXPERIMENTATION - RUN CONTROL					
RUN NUMBER	RUN START TIME	POSSIBLE RUN TERMINATION CONDITIONS			
		RUN STOP TIME	MSTOP USED IN STATEMENT MODEL	ENTITY COUNT AT TERMINATE NODE	
				STATEMENT NUMBER	RUN TERMINATED IF ENTITY COUNT AT NODE REACHES
1 OF 4	0	5000	NO	43	20
				89	(infinite)
				98	(infinite)
2 OF 4	0	5000	NO	43	20
				89	(infinite)
				98	(infinite)
3 OF 4	100	5000	NO	43	20
				89	(infinite)
				98	(infinite)

Figure 6.16 Sample Run Control template, elucidating information from the Slam II program in **Figure 2.7**

Number" field is a multi-input single-output processor that receives inputs from INIT, SIM, and GEN statements, and program topology.

The run start time and run stop time columns obtain their values from the TTSTRT and TTFIN fields of the INIT statement respectively. The processor element for generating run start time and the processor element responsible for generating run stop time are multi-input-single-output processors that take inputs from the INIT statement and program topology.

Slam II provides a method for terminating a simulation run aside from defining a stop time. A simulation can be terminated by assigning a non-zero value to the

global variable MSTOP. The value for column 4 of the run control template is obtained by scanning all occurrences of the ASSIGN statement for a reference to the global variable MSTOP. A single-input-single-output processor is used to obtain this value.

The final two template columns obtain their values from the TERMINATE statements in the network model of the program. A single-input-single-output processor whose input is the statement number field of the TERMINATE statement is used to generate the value in the Statement number column of the template. A single-input-single-output processor whose input is the termination count field (TC) of the TERMINATE statement is used to generate the value in the last column of the template. The TC field of the TERMINATE statement specifies a termination condition for the simulation run. When the number of entities passing through the node reaches the threshold value specified in the TC field the run is ended.

The analyst has the ability to specify new values for run start time and run stop time directly in the template. Template editing is discussed in depth in Chapter 7.

Identifier Initialization

The identifier initialization template in **Figure 6.17** elucidates information pertaining to the initialization of Slam II system variables.

The processor element used for calculating a value for the run number column is the same as that used for the previous program template. The processor element used to calculate a value for the identifier name column is a multi-input-single-

IDENTIFIER INITIALIZATION - BY RUN				
RUN NUMBER	IDENTIFIER NAME	ALIAS NAME	CLASSIFICATION OF IDENTIFIER	INITIAL VALUE
1 OF 4	XX(1)	TRAINCARS	VARIABLE IDENTIFIER	(0)
	XX(2)		VARIABLE IDENTIFIER	(0)
	XX(3)		VARIABLE IDENTIFIER	0
	XX(4)		RUN PARAMETER	4
	XX(5)	REPOS	CONSTANT IDENTIFIER	5
2 OF 4	XX(1)	TRAINCARS	VARIABLE IDENTIFIER	(0)
	XX(2)		VARIABLE IDENTIFIER	(0)
	XX(3)		VARIABLE IDENTIFIER	0
	XX(4)		RUN PARAMETER	5
	XX(5)	REPOS	CONSTANT IDENTIFIER	5

Figure 6.17 Sample Identifier Initialization template, elucidating information from the Slam II program in **Figure 2.7**

output processor that receives input from the INTLC statement and the program topology. The program topology is needed to determine which simulation run the VAR field of the INTLC statement belongs to.

When the template is first displayed, information is sorted by increasing run number. Sorting can also be done according to identifier name. This would allow the analyst to see, for each identifier, all run initializations pertaining to it. This is a feature common to all E/Slam templates.

The "Alias Name" field obtains its value from a single-input single-output processor whose input is the EQUIV statement. This field shows the alias name of an identifier, if one has been defined.

The identifier classification is obtained through an analysis of how the identifier is used in the program. If the identifier is assigned a value once as a pre-study activity and never updated during the simulation run then it is classified as a *constant parameter.* If the identifier is initialized prior to each run and never updated during the run then it is classified as a *run parameter.* If the identifier is assigned a value following initialization then it is classified as a *variable identifier.* The values for this column are not obtained from a specific parameter field of a statement; they are obtained by performing a high level analysis of statement positioning in the program together with usage information. The processor element used to calculate the value of this field is a multi-input-single-output processor whose input is INTLC statements, ASSIGN statements and program topology.

The processor element used to calculate the value of the "Initial Value" field is a multi-input-single-output processor that obtains its input from the INTLC statement and program topology.

6.4.6 Data Collection Templates

Figure 6.18 is an example of the "Tables and Plots" template. It is used to elucidate information about data collection and display. Information is extracted from Slam II COLCT, TIMST, STAT, RECORD and VAR statements to produce entries for the dynamic part of the template.

Data collection that applies to all simulation runs (i.e., data collection specified in the Network section of the program), is listed first in the template. This

TABLES AND PLOTS GENERATED					
RUN NUMBER	STATISTICS COLLECTED ON	DESCRIPTION	OUTPUT DISPLAYED AS	STATISTICS COLLECTED AT	
				STATEMENT	LINE NUMBER
ALL	TNOW - ATR(1)	TRAIN SVC TIME	PLOT	COLCT	42
	TNOW - ATR(2)	TRUCK SVC TIME	TABLE	COLCT	88
1 OF 4	NNQ(2) vs TNOW	TRAIN Q LEN vs TIME	PLOT	REC	103
				VAR	104
	NNQ(3) vs TNOW	CARRIER 1 Q LEN vs TIME	PLOT	REC	103
				VAR	105
	NNQ(5) vs TNOW	CARRIER 2 Q LEN vs TIME	PLOT	REC	103
				VAR	106
	NNQ(6) vs TNOW	CARRIER 3 Q LEN vs TIME	PLOT	REC	103
				VAR	107

Figure 6.18 Sample Tables and Plots template, elucidating information from the Slam II program in **Figure 2.7**

is denoted by the entry "ALL" in the Run Number field. This is followed by information pertaining to each simulation run itemized by order of increasing run number.

Data collection involving a VAR/RECORD statement pair is combined into a single template entry for clarification. The RECORD statement defines the independent variable and the VAR statement defines the dependent variable for generating output. To generate the template entry for the "Statistics Collected on" column, a 3-to-1 processor is used. The processor receives input from the INDVAR field of the RECORD statement, the DEPVAR field of the VAR

statement and program topology. Similarly, the "Description" column obtains its value from a 3-to-1 processor. Input is received from the ID field of the RECORD statement, the ID field of the VAR statement, and program topology.

6.4.7 Usages Templates

Figure 6.19 is an example of the identifier usage template. This template is used to elucidate information on the usage of the XX() and II identifiers in the program. This template elucidates information extracted from the Slam II INIT and EQUIV control statements and any network statement that allows an identifier to be specified in its parameter fields.

Processor elements for generating usage template values do not receive input from program topology. This is because there is no notion of run number in these templates. Information is elucidated on the usage of program resource elements through the entire scope of the program.

IDENTIFIER USAGE					
IDENTIFIER	ALIAS	CLASSIFICATION	DESCRIPTION OF USAGE	USED AT	
				STATEMENT	LINE NUMBER
XX(1)	TRAINCARS	VARIABLE IDENTIFIER	ASSIGNED A VALUE	ASSIGN	24
			ACTIVITY CONDITION	ACTIVITY	29
			ACTIVITY CONDITION	ACTIVITY	30
			ACTIVITY CONDITION	ACTIVITY	31
			R-VALUE	ASSIGN	32
XX(2)		VARIABLE IDENTIFIER	ASSIGNED A VALUE	ASSIGN	26
			RESOURCE COUNT ALTERATION	ALTER	27
			ASSIGNED A VALUE	ASSIGN	32
			RESOURCE COUNT ALTERATION	ALTER	33

Figure 6.19 Sample Identifier Usage template, elucidating information from the Slam II program in **Figure 2.7**

7 Editing with E/Slam Templates and Program Generation

"The art of progress is to preserve order amid change and to preserve change amid order"—Whitehead

This Chapter discusses E/Slam's update phase and program generation phase. The update phase (or edit phase), provides an environment for reliable editing, and is used for specifying new simulation experiments in a way that does not require the analyst to interact directly with the program source code. The generation phase produces an up to date version of the program source code reflecting all changes made to the internal model during the update phase.

To say that a software environment supports editing, means it has the functionality in its user interface component and other software components, to instrument a change in program specification. The author would like to stress that the ability to merely support basic text edits is not enough. The ultimate goal should be to provide an environment that supports *reliable* editing. Reliable editing entails providing an environment that ensures whatever program changes are made are acceptable ones (i.e., they do not introduce errors or inconsistencies in program specification). This can be realized by designing the editing environment to posses syntactic and semantic knowledge about the subject programming language. The system could tap into this knowledge to verify the acceptability of program updates instrumented by the user.

Significant strides can be made in expanding the size of audience that can specify and run new experiments on simulation models if the software environment they interact with combines reliable editing with a user interface that does not require them to interact directly with the program source code. This is achieved by placing a program understanding system between the program source code and the user interface. The program understanding system has built-in knowledge about the syntax and semantics of the programming language supported by the environment. Thus, it has the ability to recognize pertinent information, extract the information, and elucidate the information while leaving the programming language syntactic structures —and the complexity associated with understanding them— behind.

Figure 7.1 distinguishes three levels of editing ability. The approach shown in part a) demonstrates the traditional way in which changes are made to a program. If we call the program source code **P**, the user makes changes to a verbatim image of **P** through the user interface of a text editor. To update a program properly using this method, the analyst must have an understanding of the programming language used. A second characteristic of this method is the editor allows the analyst to make any change whatsoever to the program—even if the change causes the program to be in an erroneous or inconsistent state.

The approach shown in **Figure 7.1 b)** demonstrates knowledge-based editing of program **P**. Here the user makes changes to the program indirectly through a user interface that displays relevant program information unaccompanied by programming language syntax. What the analyst sees here, is not the program source code, but the product of the program understanding system's detailed analysis

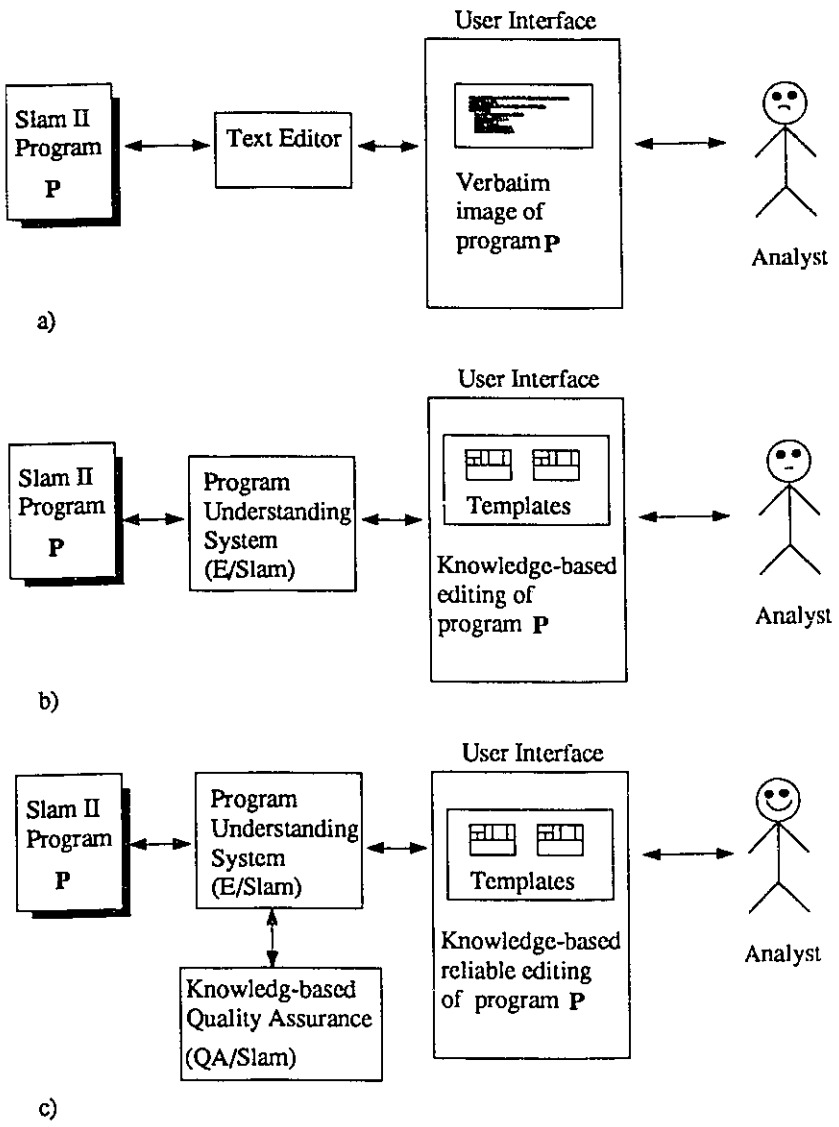


Figure 7.1 Three approaches for modifying program source code:
a) text editing
b) Knowledge-based editing
c) Knowledge-based reliable editing

of P. When the analyst initiates a program update, the program understanding system handles the task of instrumenting the specified changes at the appropriate location in the source code.

E/Slam supports knowledge-based editing through its documentation templates. The analyst makes updates to the program through the same templates used to elucidate program information. E/Slam, through its understanding of Slam II programs knows where to update the program source code to reflect parametric changes instrumented by the analyst. E/Slam also knows immediately upon entry of a new value, whether or not it is an acceptable value. A value is deemed unacceptable if its presence causes the program to be syntactically incorrect. When a value is unacceptable, E/Slam refuses to instrument the change in its internal model of the Slam II program, and notifies the analyst immediately, so that a correct value can be specified.

The approach shown in **Figure 7.1 c)** takes things one step further than b) by offering built in quality assurance. The same features offered in part b) exist in c). However, the validation of the update process is enhanced by a knowledge-based quality assurance module. This module has intelligence about what constitutes inconsistencies in program specification (i.e., semantic errors). This expertise is drawn upon in the process of making program updates. Programming inconsistencies are discussed in detail in Section 8.1.

QA/Slam is a knowledge-based quality assurance module implemented in Prolog, and has a rule-base of knowledge about potential sources of inconsistency in Slam II program specification. QA/Slam applies its rules to a fact base of information about the Slam II program at hand. The results of its findings are

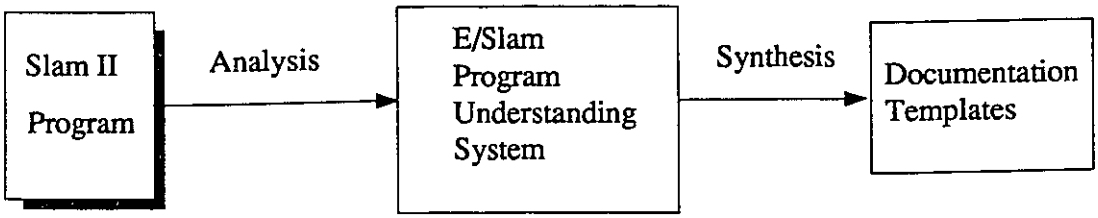
displayed in a window for the analyst to peruse, accompanied by advice on how to alleviate the inconsistencies. If no inconsistencies are discovered then the program is given a certification of correctness.

Figure 7.2 demonstrates the major distinction between using a E/Slam for elucidation and using E/Slam to update simulation experiments. **Figure 7.2 a)** demonstrates the forward engineering approach. Here we subject the program source code to an analysis phase, followed by a synthesis phase in order to produce documentation templates. When we reverse this process by taking updates from the documentation templates and input them to a program generator as shown in **Figure 7.2 b)**, we have a process for updating experiments in Slam II programs that removes the analyst from direct interaction with the program source code.

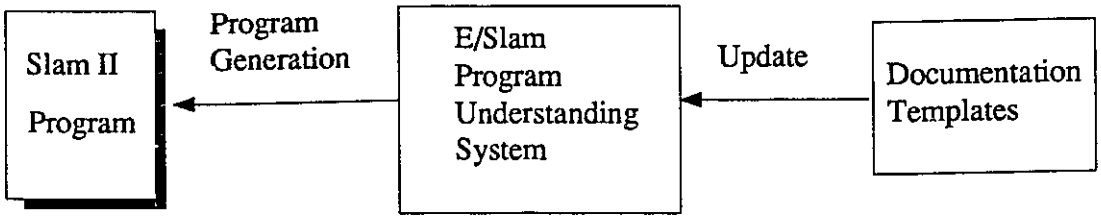
Figure 7.3 lists the major steps involved in implementing the features shown in **Figure 7.2 b)**. In the figure, modules are represented by boxes without shadows. Internal data stores are represented by a shadowed box, and processing steps are depicted as numbers enclosed by circles.

The update module is central to the configuration. It coordinates the other processes that take place. If the system is going to allow program updates, the user interface must have the facilities necessary to support the change. This includes menu options, window, etc., for the user to interact with.

In E/Slam, a change is initiated by placing the cursor in the template cell to be modified, and the menu item "Edit Template" is selected with the mouse. When the user initiates the change, the update module is notified (step 1). Accompanying the notification, is information about where in the user interface the change is



a)



b)

Figure 7.2 Two directions of E/Slam processing
a) Forward engineering: for understanding Slam II programs
b) Reverse engineering: for knowledge-based editing of the simulation experiments in Slam II programs

requested. The information supplied includes the template identifier, line number and column number of the cursor positioned in the dynamic part of the template.

Step 2 involves identifying the link between the selected entry in the user interface and the location in the internal model from which that entry was extracted. At template generation time, this information is placed in a cross reference table so the backward links can be exploited when required. E/Slam

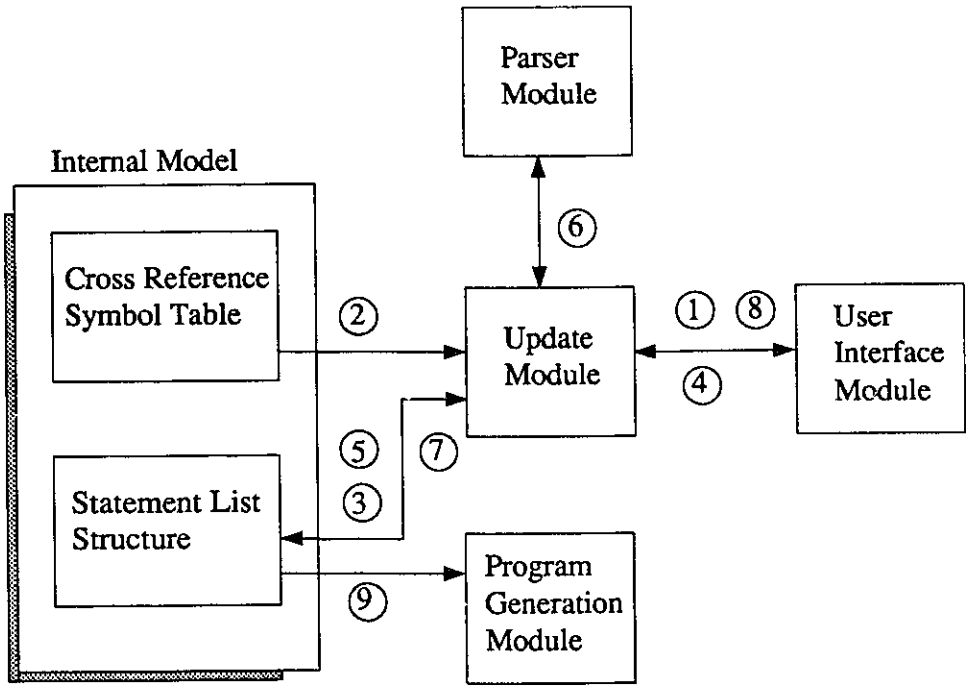


Figure 7.3 Steps involved for E/Slam to support knowledge-based editing of the simulation experiments in Slam II programs

uses the template id, line number and column number as the search key for this data. Upon completion of the search, the Update module has a handle on the information sources. The information sources are somewhere in the Statement List structure. The information sources are tapped (step 3), and the value of the selected template field is determined and displayed in a pop-up edit window (step 4). The edit window shows the current value of the template field and provides a second field for specifying its new value.

Once the new value is entered and the analyst hits the enter key, the handle is used once again to extract all internal model information necessary to reconstruct the statement that is to take on the new value (step 5). The statement is

reconstructed to the syntactic form it had in the Slam II source file, with the exception of the field that is to take on the new value. This field is specified with the new value in place of the original value.

Now that the statement is constructed, it is passed to the parser module (step 6). The parser does a syntactic analysis on the full statement and returns its result to the Update module. If the syntax is incorrect the user is notified and given the opportunity to enter another value or quit the edit process. If another value is specified and the syntax is correct the internal model is updated to reflect the new value (step 7).

Once the new value is accepted, the user interface is updated to reflect the new state of the internal model (step 8). The user interface is updated in three places: the program source code displayed in the Listslam window, the template in which the change was initiated, and any other open templates that obtain their values from the program component that was modified.

Once the analyst has completed all changes, and verified the acceptability of them using QA/Slam a new source code listing is generated that reflects the updates made to the internal model (step 9).

It should be noted that a one-to-one correspondence does not necessarily exist between a value in the internal model and a documentation template field. Data at a specific location in the internal model could be referenced to generate more than one type of template. For example, the TTBEG field of the INIT statement is referenced to generate the INIT template and it is also referenced to generate the "Parameters of Experimentation-Run Control" template. The implication of this is that if a change is made to TTBEG via either of the two mentioned templates,

and both templates are displayed on screen, then both templates must be updated to reflect the new value. E/Slam handles this using a dependency table that lists for each documentation template x, all other templates that could possibly become out dated if a change is made to the dynamic part of x.

Although controlled editing eliminates the possibility of syntactic errors, there still exists the possibility that semantic errors will contaminate the program. The consistency checking feature of QA/Slam is used to ensure a program change has not introduced an incompatibility between the changed program element and other related program elements. The concept of consistency checking and the kinds of consistency checks that can be carried out by E/Slam are discussed in Chapter 8.

8 An Embedded Rule-Based Approach for Quality Assurance

This chapter begins by identifying the sources from which Slam II program inconsistencies arise. The objective is to show how these sources of inconsistency are manifested in Slam II programs and how they act as an opposing force in our effort to produce quality program specifications. By identifying these sources of inconsistency, the reader will see how vulnerable software can be to program changes even if those changes are minor ones. The discussion continues with a demonstration of how these problems can be addressed by designing intelligent software. This will open the door to a detailed discussion of QA/Slam. QA/Slam is an embedded knowledge-base system for quality assurance (QA) of Slam II programs. We begin by identifying the sources of inconsistency in Slam II programs (Section 8.1), and then offer some background information on rule-base systems (Section 8.2).

8.1 Sources of Inconsistency in Slam II Programs

Any change made to a segment of program code, regardless of how minor it is, has the potential to put the overall system in an inconsistent state. In Slam II for example, even though a change may be localized to one parameter field, there are often hidden interdependencies between statement fields. This can impose a series of hidden constraints on the acceptable values for a field. When these constraints are violated an inconsistency in program specification (IPS) exists.

When an IPS exists between Slam II statement parameter fields the cause could be an out of range condition. This occurs when the value in a statement field limits the value of another field by acting as an upper or lower bound for the field. In Slam II this occurrence is not limited only to numeric constants specified in statement parameter fields, but is also applicable to numeric constants specified as indexes to system variables used in statement parameter fields. An IPS could also be caused by conflicting modes of use of statement fields, where one field implies something, while another field supplies contradictory information.

An IPS could also involve program topology: the relative ordering of statements in the program. Each Slam II Network statement has a graphic symbol associated with it. Sequences of network statements produce a network graph whose topology is dictated by the relative ordering of statements in the program. An IPS could arise if the topology is incompatible with values specified in statement parameter field.

An IPS also exists if a program field value is inconsistent with respect to the Slam II supporting system. The supporting system encompasses the Slam II language definition and any constraints imposed at language implementation time and installation time. In Slam II when an IPS involving the supporting system, it does not necessarily cause a syntax error. This is especially true for constraints imposed at installation time.

When an IPS exists, the conflicting elements may be local to a single statement or span across multiple statements. Furthermore there could be greater than two elements contributing to an inconsistent state. It is conceivable for an IPS to be the result of two statement fields and program topology. The following paragraphs

offer specific examples where the types of inconsistencies mentioned above could arise in Slam II.

An example of a source of inconsistency in program specification involving an out of range condition, where the conflicting elements are local to one statement, can be found in the Slam II QUEUE statement. In this statement the value specified in the initial queue content field (IQ) is bounded by the value specified in the statement's queue capacity field (QC). The IQ field indicates the number of entities residing in the queue at the start of a simulation run. The QC field indicates the maximum number of entities that can coexist in the queue. The relationship

$$IQ \leq QC$$

must hold for every QUEUE statement. If this inequality is violated then the IQ and QC fields are inconsistent with respect to each other.

A source of inconsistency in program specification where the conflicting elements are local to a single statement and the conflict belongs to the "mode of usage" category is found in the Slam II CREATE statement. The TBC field of the CREATE statement specifies the time between creations of entities emanating from the CREATE node. The MC field indicates the maximum number of entities to be created during the course of a simulation run. If the TBC field is defaulted and the MC field contains an explicit value other than 1, then an IPS exists. This is because a defaulted TBC field indicates that the time between entity creations is infinite, which means only one entity is created at the node. If the MC field contains a value greater than 1 this implies that more than one entity is expected to be created.

An example of an IPS that spans across statements would be one involving the MFIL field of the LIMITS statement and the IFL field of QUEUE statement. The MFIL field specifies an upper bound on the value that can be expressed in the IFL field of a QUEUE statement. if the condition

$$IFL \leq MFIL$$

does not hold, then we have an inconsistency in program specification.

A "mode-of-usage" IPS can arise between the AWAIT statement and RESOURCE statement. The inconsistency involves three statement fields: the IFL and RES fields of an AWAIT statement, and the IFL field of a RESOURCE statement. The RES field names a resource to be seized by entities arriving to the AWAIT node. The IFL field of the AWAIT statement indicates the file number where entities are to wait for the availability of the resource specified in the RES field. The IFL field of a RESOURCE statement indicates the file number for holding entities associated with the resource named in the RLBL field of the RESOURCE statement. An inconsistency exists if the RES field of an AWAIT statement contains a resource label *r*, while the IFL field contains an integer constant *i*, and *i* is not listed in the IFL field of the resource statement defining *r*.

These are just some of the many IPS sources present in the Slam II language. Whenever an update is made to a Slam II program, those making the change must be cognizant of these sources of inconsistency. The problem is there is no guarantee that the person making the update will be cognizant of all interdependencies at all times. The disadvantage of this approach is that quality is neither consistent nor guaranteed. To be able to make the assertion that a piece

of software is of a high quality, QA principles must be enforced not only during the initial engineering of the software, but also during the maintenance phase of the software development lifecycle. To ensure a certain degree of quality is met consistently for all program updates, tools that offer an environment for updating programs should also be engineered with built-in quality assurance.

In E/Slam's case, the need for built-in quality assurance is two fold. E/Slam is a tool intended for analysts who wish to experiment with simulation programs, yet do not necessarily have an intermediate or advanced understanding of the language. Thus, it is not recommended that the onus be placed entirely on the analyst to identify the IPS's that could arise. Especially since one typically becomes knowledgable of these sources inconsistency after using the language for quite some time. This is partially because many of the sources of inconsistency are undocumented.

8.2 Rule-Based Systems

Software systems can be classified according to their domain of intended use. Any software system of at least moderate complexity has knowledge specific to its problem domain encoded within itself in some form or another. This knowledge could either be hard coded into the system or be specified in a more autonomous way. A rule-based system is a software system that has its application domain knowledge specified in the form of production rules that are independent of the rest of the system. There are three main components of a rule-based system: a knowledge-base, working memory, and an inferencing mechanism (Golshani 1990).

The knowledge-base is the holding place for the systems domain knowledge. This knowledge is encoded as rules and facts. A rule is an expression of the form

$$\textbf{IF } < cond > \textbf{ THEN } < action >$$

where <*cond*> is the rule antecedent and <*action*> is the rule consequent. Each side of the rule can be expressed as a conjunction of predicates. All predicates in the antecedent must evaluate to TRUE, in order for the rule consequent to be asserted. The facts in the knowledge-base are general assertions about the application domain. Facts can be conceptualized as rules whose antecedent is TRUE:

$$\textbf{IF } TRUE \textbf{ THEN } < action >$$

In this case, the rule action represents the fact to be stated.

For rule-base systems to offer meaningful conclusions, they must be able to apply general knowledge contained in their knowledge-base to particular contexts or situations. This means they need access to factual information related to the problem at hand. This information can be supplied before the system starts inferencing, or it can be requested from the environment during system operation. The environment in this case could be a human interacting with the system, or hardware sensors. Information supplied prior to inferencing is stored in a *fact-base*. The knowledge-base taken in combination with the fact base is called the *context*, as shown in **Figure 8.1**

Working memory is an extension to the fact-base. The difference between a fact-base and working memory is one of permanency. General knowledge is stored in the knowledge-base. Knowledge specific to the current problem is

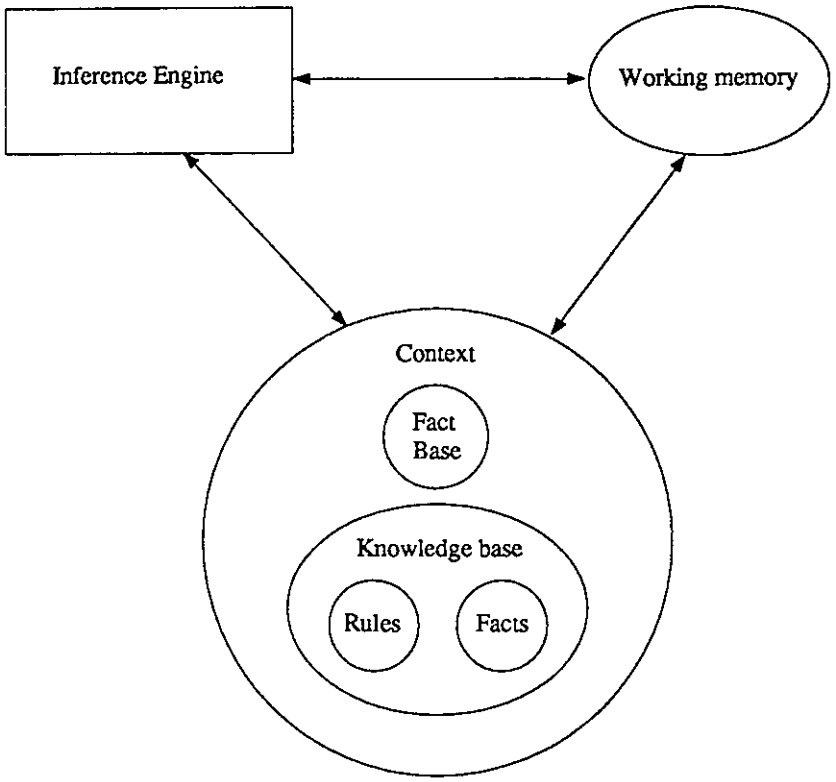


Figure 8.1 The major components of a rule-based system

stored in the fact base. Working memory holds intermediate knowledge and conclusions brought about through inference. Prior to inference engine execution the workspace is empty.

It is the inference engine's responsibility to inspect the rules in the rule-base and determine which ones are *firable* given the factual information at hand. A rule is firable if its antecedent evaluates to TRUE. Amongst those rules whose antecedent evaluates to TRUE, the inference engine must select one for *firing*.

A rule is said to *fire* when its consequent is evaluated. Rule evaluation has the effect of changing the system state by updating working memory. As the system state changes, rules whose premises previously did not evaluate to TRUE, may now evaluate to TRUE, making them eligible for firing. This cyclical pattern of rule selection, rule firing, and working memory update continues until eventually a final system state is reached. When no rules are firable this is usually the testimony of a final state.

A rule-based system is a knowledge-based system. When an RBS is embedded within a conventional software system, the total package is called an *embedded knowledge-based system* (EKBS) or more specifically an *embedded rule-based system* (ERBS). What we mean by a conventional software system is one whose architecture is such that the system does not belong to the class of artificial intelligence systems. The embedded feature serves to enhance the functionality of the over all system by offering built-in expertise.

The value of software systems can be enhanced substantially if designed as EKBS's. Systems that support the development and modification of computer programs for example, can be enhanced with a KBS that identifies sources of inconsistency in program specification. It is often not necessary for the KBS component to be a full fledged expert system, and an EKBS should not be evaluated according to how good of an expert system it is. Instead, it should be judged according be the degree in which the conventional component is enhanced by the presence of the AI component.

8.3 QA Phase in E/Slam

Knowledge-based systems (KBS) are an ideal candidate for handling the quality assurance issues related to program modification. With an embedded KBS one could build a knowledge-base of rules that check for IPS's that arise in Slam II programs. Following a program change made by the analyst, the KBS could be activated to locate all inconsistencies in program specification. This type of system would notify the analyst of any inconsistencies discovered and would offer suggestions about how to correct the problem. This value added approach to software engineering allows the analyst to concentrate more on solving real world problems using simulation rather than on the programming language used to carry out the simulation.

QA/Slam is E/Slam's embedded knowledge-based system written in Quintus Prolog. QA/Slam has a rule-base of knowledge for identifying inconsistencies in program specification. Problem diagnosis and suggested fixes are provided by QA/Slam for each IPS it discovers.

The QA phase of E/Slam is implemented as an ERBS. Its purpose is to offer assurance that programs updated in E/Slam maintain a respectable level of quality unhindered by program updates. **Figure 8.2** shows where the QA phase fits in relation to the rest of E/Slam.

During the QA phase of operation, E/Slam's internal model is consulted for information about the Slam II program. A rule-based system with knowledge about Slam II program inconsistencies studies the Slam II program information supplied by the internal model and produces a QA report that documents its findings. All inconsistencies discovered in the Slam II program are documented

in detail in the QA report. Each report entry offers an explanation of the cause of an inconsistency, the location of elements contributing to the inconsistency, and suggestions on how the problem can be corrected. The QA phase consists of two main modules: QA/Slam preprocessor and QA/Slam; each is defined in the following sections.

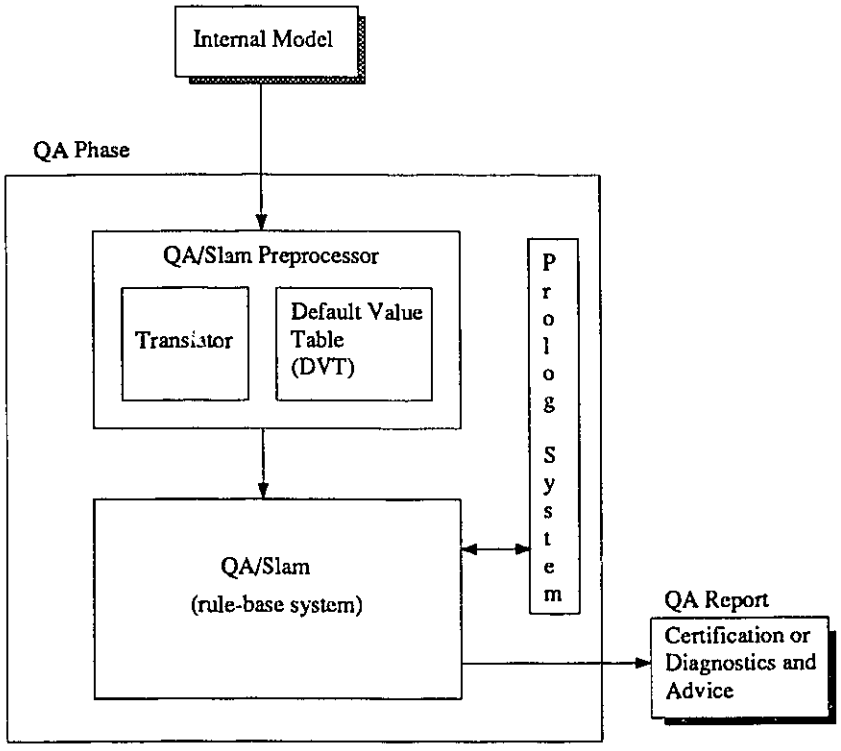


Figure 8.2 Architecture of E/Slam's QA phase

8.4 QA/Slam Preprocessor

The QA/Slam preprocessor is module that translates Slam II program information extracted from E/Slam's internal model, into a series of Prolog facts intelligible by QA/Slam. Extracted from the internal model is information about each Slam II statement, information about network graph connectivity and information about simulation run specifications.

The preprocessor is necessary because E/Slam and QA/Slam are separate run time modules. E/Slam is written in C and QA/Slam is written in Prolog. Because QA/Slam does not have direct access to E/Slam's internal model, the information held in the internal model must be placed in a fact base.

The QA/Slam preprocessor is written in C and is physically part of the E/Slam run time system. Functionally speaking however, it is part of QA/Slam, since its sole purpose is to make information held in the internal model accessible to the rule-based system.

As part of the translation step, the preprocessor references a Slam II statement default value table (DVT). This table contains the language defined default value for each Slam II statement field. During the translation process, whenever a defaulted statement field is encountered in the internal model, the value for that field is looked up in the DVT and included in the Prolog fact that describes the statement.

The output of the preprocessing stage is saved in an ASCII text file which serves as input to QA/Slam. The text file contains a series of Prolog facts about

the Slam II program. There are three types of fact: statement facts, connectivity facts and run facts. Each is described in detail in the following sections.

8.5 QA/Slam

QA/Slam is an embedded rule-based system for quality assurance of Slam II programs. QA/Slam inspects the sources of inconsistency in programs to identify conflicting states, and offers guidance on how to correct the inconsistencies discovered.

For a human to identify many of the inconsistencies identified by QA/Slam would require an in depth understanding of the Slam II language. It is not acceptable to require the user to posses this knowledge since E/Slam is meant to be used by those who are not entirely familiar with Slam II. Consequently, the knowledge required to identify programming inconsistencies is expressed in a knowledge-base that is consulted by QA/Slam.

The analyst initiates the QA phase through the E/Slam user interface via a panel button. This invokes the QA/Slam preprocessor which references the internal model of the Slam II program and extracts relevant facts about the program. This information is formatted and placed in a file. Control is then passed to the QA/Slam process which inputs this file as its fact base. At this time, QA/Slam also loads its knowledge-base file and its diagnosis file. The diagnosis file contains natural language text that is used to document inconsistencies. While the inference engine operates, it applies general QA rules in the rule-base to the specific program facts in the fact-base. As inconsistencies are discovered, diagnostic information identifying the nature and location of the problem together

with suggested remedies for the problem, are written to the QA report file. Once all eligible rules have fired, the inference engine concludes its execution and control is passed back to the E/Slam process. Once E/Slam regains control it opens the QA report file and displays it in a text window for the analyst to view.

When consistency checking is invoked, it is carried out on the current state of the internal model. Thus, the user can perform checks either on the original program, or on a modified version of the program, depending upon whether the update phase has been entered prior to invoking the QA phase.

Consistency checks are initiated by the user instead of being carried out automatically following each change to the internal model to prevent unwanted IPS messages. When the user initiates a program update that will involve changing values in more than one location in the program, it would be annoying for the analyst to have the IKBS notify him of IPS's that are simply caused by his being in the middle of a group of field updates.

As shown in **Figure 8.3**, QA/Slam is comprised of four components: fact-base, rule-base, diagnostics file, and QA/Slam inference engine. These components run on top of Quintus Prolog to produce a QA report. Each component is discussed in depth in the following sections.

8.5.1 Fact Base

The fact base supplies QA/Slam with information specific to the Slam II program under study. As shown in **Figure 8.3**, three kinds of facts reside in the fact-base: statement facts, connectivity facts, and run facts. With *statement* facts describing statement field values, *connect* facts describing statement topology, and

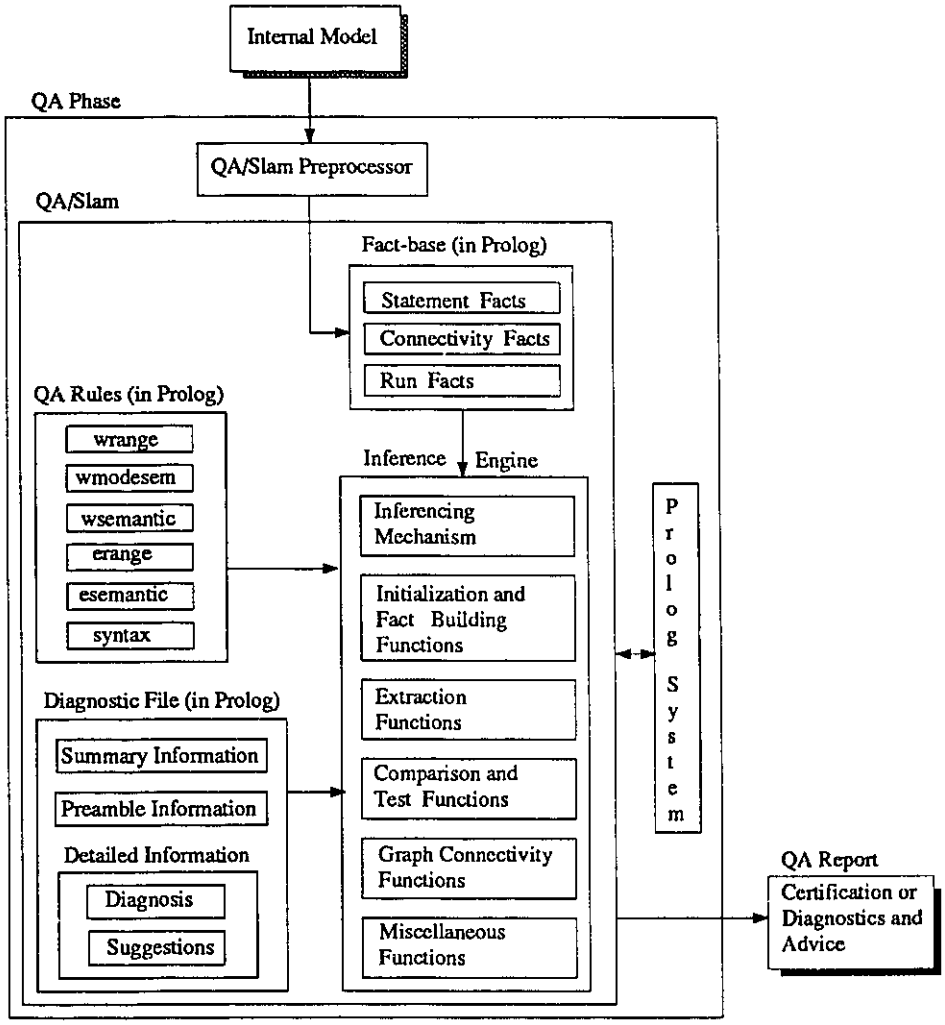


Figure 8.3 Components of QA/Slam

run facts supplying run information, all the necessary Slam II program information is available for QA/Slam to perform its function.

Statement Facts

Statement facts are used to describe the content of Slam II statements in the simulation program. A one-to-one correspondence exists between statements in the program and statement facts in the fact-base. A BNF representation of statement fact structure is shown in **Figure 8.4**. In the following BNF diagrams entries enclosed by '< >' denote non-terminal symbols. Entries enclosed by '{}' indicate zero or more occurrences of the enclosed symbols, and '{ }^{x}'denotes an upper bound of x repetitions of the enclosed symbols. The 'I' character denotes the selection operation.

Knowledge about an individual Slam II statement is expressed as a Prolog structure with a functor name of **statement**. Each **statement** predicate or **statement** fact, contains an Slbl field, Snum field, Sid field, and a Sparm list.

The Slbl field holds the statement label associated with the Slam II statement. The Snum field holds the line number at which the statement occurs in the program. The Sid field contains the Slam II keyword for the statement name (e.g., CREATE, INIT, etc.).

All program statements in Slam II follow a similar syntactic format consisting of a Slam II statement keyword followed by a sequence of parameter fields. Information about an individual Slam II parameter field is expressed in a Prolog list of the form [Fid,Foff,[DeflFval]]. This list is called a parameter value packet

<Statement Fact> ::= statement('<Slbl> ', <Snum> , '<Sid> ', [<Sparm>]).

<Slbl> ::= <Letter> {<Letter> | <Digit> }^{3}

<Snum> ::= <Digit> | {<Digit>}

<Sid> ::= ACCUMULATE | ACTIVITY | .. | VAR

<Sparm> ::= {<Pvp>}

<Pvp> ::= [<Fid> , <Foff> , [<Fval>]] | <Pvp> , <Pvp>

<Fval> ::= [<Def> , <Val>]

<Def> ::= d | e

<Val> ::= <Token> | <Token> , <Val>

where:

Statement Fact	Information about a single Slam II statement
Slbl:	statement label
Snum:	statement line number
Sid:	statement keyword
Pvp:	parameter value packet
Sparm:	statement parameter field
Fid:	mnemonic field identifier of a statement
Foff:	holds the parameter field offset within a Slam II statement
Fval:	Slam II statement field information
Def:	default flag for a parameter field (d = default, e = explicit specification)
Val	Slam II statement field value
Digit	A number between 0 and 9
Letter	Any upper case letter of the alphabet

Figure 8.4 BNF for QA/Slam statement fact

and is represented by the non-terminal <Pvp> in **Figure 8.4.** If a Slam II statement consists of *n* parameter fields then its associated statement fact contains *n* parameter value packets—one for each field.

The Fid field of a parameter value packet is the mnemonic field identifier of the Slam II statement (e.g., IFL, TBC, etc.). Foff is the parameter field offset within the Slam II statement, where the first field—the field next to the statement keyword—has a Foff value of one.

The third list element in a Pvp is a Prolog list. Its content is denoted by the non-terminal <Fval>. The <Def> field of <Fval> is the default flag for the Slam II parameter field. It contains one of two possible values: *d* or *e*. The value *d* indicates the field is defaulted in the Slam II program, while the value *e* indicates the field value is explicitly specified.

The statement field value is specified in the list tail denoted by <Val>. The statement parameter field value is expressed here even if the field is defaulted in the Slam II program. Each element in the list <Val> is a Slam II language token. The example in **Figure 8.5** shows a Slam II CREATE statement and its equivalent QA/Slam **statement** fact.

Connectivity Facts

The connectivity facts in the fact base describe the node connectivity of the Slam II network model. Each Slam II network statement has a Slam II network symbol associated with it. When the network statements are taken as a group their network symbols form a network graph representation of the simulation program.

```
CREATE , TBC , TF ,  MA , MC , M ;

                a)

5  LA1    CREATE, EXPON(12) ,     , 1     ,     ,    ;

                b)

statement( 'LA1' , 5 , 'CREATE' ,
        [ [ 'TBC' , 1 , [ e , 'EXPON' , '(' , '12' , ')' ] ] ,
          [ 'TF'   , 2 , [ d , '0'] ] ,
          [ 'MA'   , 3 , [ e , '1' ] ],
          [ 'MC'   , 4 , [ d , 'infinity' ] ] ,
          [ 'M'    , 5 , [ d , 'infinity' ] ]
        ]).

                c)
```

Figure 8.5 a) General format of the Slam II CREATE statement
b) A typical Slam II CREATE statement.
c) The Slam II statement from b) encoded as a QA/Slam **statement** fact

Rules in the knowledge-base require access to this connectivity information to identify program inconsistencies that relate to network topology.

Knowledge about Slam II connectivity is expressed using a **connect** predicate. Each **connect** predicate or **connect** fact, contains a SrcNum field, ArcType field and a SinkNum field as shown in **Figure 8.6.**

Each network statement in the program that routes to another network statement has its connection described by a connect fact. If a network statement has multiple emanating arcs, then each arc is described by a separate fact.

<ConnectFact>	::=	connect(<SrcNum> , <ArcType> , <SinkNum>).
<SrcNum>	::=	*positive integer*
<SinkNum>	::=	*positive integer*
<ArcType>	::=	n | b

where:

ConnectFact	Connect fact
SrcNum:	Statement number from which arc emanates
SinkNum:	Statement number at which arc terminates
ArcType:	Arc type (n = normal , b = balk)

Figure 8.6 BNF for the QA/Slam **connect** fact

The SrcNum field represents the statement number of the source node, SinkNum represents the statement number of the sink node, and ArcType describes the type of arc that connects the source node to the sink node.

ArcType can be one of two types: balk or normal. Balk arks represent connectivity expressed through the use of BALK() parameters in Slam II statements. The only statements that can have this kind of arc emanating from them are AWAIT, PREEMPT, and QUEUE statements. Balk connectivity is expressed by placing *b* in the ArcType field.

Normal arcs are defined either implicitly by the ordering of network statements in the program, or explicitly through the use of ACTIVITY statements. When two statements appear in succession an arc is assumed to connect their network symbols in the network graph. The ACTIVITY statement allows one to place

<RunFact> ::= run(<Snum> , <Rnum>).

<Snum> ::= 1 | 2 | 3 | ...

<Rnum> ::= all | 1 | 2 | ...

where:

RunFact:	a statement of simulation run information
Snum:	statement number
Rnum:	run number with which statement is associated (all = statement applies to all runs)

Figure 8.7 BNF for the QA/Slam **run** fact

an explicit arc between network statements. Normal connections are the most common connections found in Slam II programs, and are denoted by the value *n* in the ArcType field.

Run Facts

Knowledge related to Slam II simulation runs is expressed using the QA/Slam **run** predicate. The **run** predicate or **run** fact, is used to describe the association between program statements and simulation runs. The BNF for this predicate is shown in **Figure 8.7**. The non-terminal <Snum> represents the program statement number. The non-terminal <Rnum> represents the run number in which the statement comes into effect.

Generally speaking, if a statement comes into affect for run number *i* then it stays in effect until the end of the simulation study, or until the next occurrence of the same statement type. An exception to this rule occurs when dealing with variable initialization statements such as INTLC. Since one is able to initialize more than one variable in a single statement, the situation can arise where the scope of applicability is different for each variable expressed. This would occur if one variable in the INTLC statement is reinitialized in a later run while another variable in the INTLC statement is not.

The Slam II supporting system requires that network statements apply universally to all runs. This is expressed by placing the atom 'all' in the run number field of each network statement's **run** fact. Since this principle does not apply to control statements, the run number field for these statements always contains an integer value.

Because the number of simulation runs to which a control statement applies is a function of the statements that appear after it in the program, one interprets the **run** predicate as a description of the earliest run number with which a statement can be associated.

8.5.2 QA Rules

General knowledge about sources of inconsistency in Slam II programs is stored as quality assurance (QA) rules in the rule-base. A QA rule is a condition-action pair that states the conditions that must hold for the associated action to take place. An action either places an assertion about the nature of the Slam II

program in working memory, or initiates documentation of the inconsistency. The QA/Slam rule–base contains 66 rules specified in Prolog.

Format of Rules

A QA rule is a pair of Prolog lists separated by the '==>' operator. Elements in the list positioned before the '==>' operator are rule premises. Elements in the list following the '==>' operator are rule actions. The '==>' operator represents the IF-THEN relation described in Section 8.2.

The BNF specification for a QA rule is shown in **Figure 8.8**. Every QA/Slam rule begins with the operator symbol **rule**, followed by a name, a list of rule premises, and a list of rule actions or consequents. These three fields are denoted by <rule_name>, <premise_list>, and <consequent_list> respectively.

Rule Names

Each rule is assigned a unique name for identification purposes. The rule name is expressed as a concatenation of substrings that reveal information about the nature of the rule. The substrings from which the name is constructed are denoted by non-terminals <Rtype>, <Num>, and <Auxiliary>.

The rule type (<Rtype>) identifies the severity and type of program inconsistency identified by the rule. If the first letter of the substring is *w*, the rule identifies a warning condition. If the first letter is an *e*, the rule identifies an error condition. The remaining letters express the classification of the inconsistency. The two main classifications are *syntax* exception, and *semantic* exception.

<QASlamRule>	<RuleName># [<PremiseList>] ==> [<ConsequentList>]
<RuleName>	<Rtype> <Num> | <Rtype> <Num> <Auxiliary>
<PremiseList>	<Num>#<Premise> | <PremiseList> , <PremiseList>
<ConsequentList>	<Action> | <ConsequentList> , <ConsequentList>
<Rtype>	wrange | erange | wmodesem | emodesem | wsemantic | esemantic | syntax
<Num>	<Digit>{<Digit}>
<Digit>	0 | 1 | 2 | ... | 9
<Action>	*a Prolog predicate*
<Premise>	*a Prolog predicate*
<Auxiliary>	AUX<Num>

where:

QASlamRule	QASlam quality assurance rule
RuleName	unique rule identifier
PremiseList	a list of predicates that constitute the antecedent of the QA rule
ConsequentList	a list of predicates that constitute the rule action
Rtype	an identification of the category with which the rule belongs
Num	an integer value
Action	a rule consequent predicate
Premise	an antecedent predicate
Auxiliary	a rule name extension used to identify information gathering rules

Figure 8.8 BNF for QA/Slam knowledge-base rule

Semantic exceptions can be further subdivided into *range* exceptions and *mode* exceptions. Range exceptions and mode exceptions are discussed in Section 8.2.

Concatenated with <Rtype> is a sequence number <Num>. Sequence numbers are used to distinguish rules of the same type.

The rule-base contains two types of rules: *diagnostic* rules and *information-gathering* rules. Information-gathering rules are distinguished from diagnostic rules by an auxiliary extension <Auxiliary> in the rule name. The extension is simply the substring *aux*, followed by a sequence number. Because, each information-gathering rule is associated with a single diagnostic rule, the diagnostic rule name is prefixed to the information-gathering rule name. Sequence numbers are appended to the auxiliary portion to distinguish information-gathering rules associated with the same diagnostic rule. Diagnostic rules and information-gathering rules are discussed in section 8.4.2.2.

Rule Premise

The rule premise list <PremisList> is a list of Prolog predicates that specify the preconditions that must be met for the rule consequent to be assertable. All rule premises are placed in the list delimited by '[' and ']' preceding the '==>' operator. A number is associated with each premise to facilitate easier reading; this is purely for aesthetic reasons.

A single premise is denoted by the token <premise> in **Figure 8.8**. A premise is either a Prolog defined predicate or a QA/Slam defined predicate. A QA/Slam defined predicate can be an extractor predicate, comparison and test predicate,

graph connectivity predicate, or a miscellaneous predicate. These are described in section 8.5.4.

Rule Consequent

The rule consequent is a list of actions to be taken when the specified premises are true. The list follows the '==>' operator and its format is analogous to that of the premise list.

A rule action (<Action>) can be any valid Prolog predicate. However, in practice it is either a diagnostic predicate or an *assert* predicate. Diagnostic predicates initiate documentation of a Slam II program inconsistency. Information about the inconsistency is placed in the QA report. *Assert* is a Prolog defined predicate used to place facts in working memory.

Diagnostic Rules and Information-Gathering Rules

Diagnostic rules identify inconsistencies in program specification. Information-gathering rules draw useful conclusions about the nature of the Slam II program. These conclusions are stored in working memory for the diagnostic rules to use.

The two types of knowledge-base rules are distinguished by the affect they have on the system when the rule fires (i.e., the kind of action expressed as the rule consequent). If the rule consequent adds an entry to the QA report it is a diagnostic rule. If the consequent adds a fact to working memory it is an information-gathering rule.

Diagnostic rules are used to identify syntactic and semantic errors in Slam II programs. When this type of rule fires it causes diagnostic information about the inconsistency to be formatted and entered in the QA report, accompanied by

suggestions on how to fix the problem. The majority of rules in the knowledge-base are of this type.

When an information-gathering rule fires an assertion is made about the nature of the Slam II program. The assertion does not describe an inconsistency, it simply provides factual knowledge about the program. This knowledge is added to working memory for the diagnostic rules to use.

An extract of QA/Slam's rule-base is shown in Figures **8.9**, **8.10** and **8.11**. Two information-gathering rules (wsemantic1aux1, wsemantic1aux2) are shown accompanied by one diagnostic rule (wsemantic1). *Wsemantic1aux1* and *wsemantic1aux2* query the fact base to determine whether a constant value is assigned to the system variable STOPA. The *assertz()* predicate places the value instantiated with its second argument in working memory. Arguments enclosed by single quotes, or beginning with a lower case letter are constants. Arguments beginning with an upper case letter are variables. When a rule is evaluated the predicates in its premise list are evaluated from top to bottom (i.e., by order of increasing premise number). As the evaluations are carried out, variables are instantiated to values that are used as arguments to predicates further in the premise list. A complete list of QA/Slam rules is given in (Wendt et al. 1990).

Diagnostic rules do not produce side affects in working memory. This means there is no constraint on how they are ordered in the rule-base. Information-gathering rules on the other hand are typically associated with a small number of diagnostic rules, and must be processed before the associated diagnostic rules are processed. Premise 1 and 4 of *wsemantic1* reference facts asserted by *wsemantic1aux1* and *wsemantic1aux2*. Consequently, *wsemanticaux1* and

```
/*
IF
        STOPA(I) exists in the DUR field of an ACTIVITY statement
        (1#,2#)
AND
        STOPA exists in an ASSIGN statement as an L-value
        (3#,4#)
AND
        the associated R-value is a non-constant
        (5#,6#)
THEN
        assert the fact that STOPA is assigned the value of a
          non-constant expression

*/


rule warnModelaux1#
 [
        1# cont('DUR','ACTIVITY',Ln1,Label,_,DURvalue),
        2# argument_to('STOPA',DURvalue,DURarg),
        3# cont('VAR','ASSIGN',Ln2,_,Ofst,VARvalue),
        4# is_one_of(VARvalue,['STOPA']),
        5# cont('VALUE','ASSIGN',Ln2,_,Ofst+1,VALUEvalue),
        6# not(is_constant(VALUEvalue))
 ]
 ==>
 [
        assertz(factbase,
        factbase:fact(assignment('STOPA',non_constant_expression)))
 ].
```

Figure 8.9 Extract from QA/Slam rule-base showing information-gathering rules warnModelaux1

```
/*
IF
        an ASSIGN node contains the value STOPA in the
        VAR field (1#,2#)
AND
        the associated VALUE field contains a numeric
        constant c (3#,4#,5#)

THEN
        assert the fact that STOPA is assigned the value c
*/

rule warnModelaux2#
 [
        1# cont('VAR','ASSIGN',Ln2,_,Ofst,VARvalue),
        2# is_one_of(VARvalue,['STOPA']),
        3# cont('VALUE','ASSIGN',Ln2,_,Ofst+1,VALUEvalue),
        4# is_constant(VALUEvalue),
        5# field_value(VALUEvalue,V)
 ]
 ==>
 [
        assertz(factbase,factbase:fact(assignment('STOPA',V)))
 ].
```

Figure 8.10 Extract from QA/Slam rule-base showing information-gathering rule warnModelaux2

wsemanticaux2 must be evaluated before *wsemantic1*. This ordering is guaranteed by ensuring the two information-gathering rules appear before *wsemantic1* in the rule-base.

Types of Diagnostic Rules

There are three criteria for classifying QA/Slam's diagnostic rules. These criteria are exception type, exception severity, and scope of inconsistency. The

```
/*
IF
        STOPA is not assigned the value of a non-constant
        expression at an ASSIGN node (1#)
AND
        STOPA(I) exists in the DUR field of an ACTIVITY
        statement (2#,3#)

AND
        no fact exists that indicates that I is assigned
        to STOPA at an ASSIGN node (4#)
THEN
        activity may have an infinite duration

*/

rule warnMode1#
 [
        1# not(factbase:
                fact(assignment('STOPA',non_constant_expression))),
        2# cont('DUR','ACTIVITY',Ln1,Label,_,DURvalue),
        3# argument_to('STOPA',DURvalue,DURarg),
        4# not(factbase:fact(assignment('STOPA',DURarg)))
 ]
 ==>
 [
        type_1_exception(wsemantic,1,'WARNING',Ln1,'ACTIVITY'
                        ,DURvalue,[DURarg])
 ].
```

Figure 8.11 Extract from QA/Slam rule-base showing diagnostic rule warnMode1

term exception represents any programming practice considered to be abnormal.

The exception types are split into two main streams: *syntactic* and *semantic* as shown in **Figure 8.12**. If a programming practice violates the grammar rules of the programming language, it is a syntax exception. Syntax exceptions never pass the program parsing stage. A programming practice is a semantic exception if the program behaves in an unexpected way due to an oversight or misunderstanding

on the programmers part. The danger of this type of exception is that it can pass the program parsing stage without errors.

Criteria			Types of Rules
Syntax	Error		syntax
Semantic	Warning	Range	warnRange
		Mode	warnMode
	Error	Range	errorRange
		Mode	errorMode

Figure 8.12 Classification of rules in QA/Slam's knowledge-base

Syntax exceptions identify coding errors that must be corrected before a Slam II program can be compiled to produce an executable program. The rule-base currently holds eleven rules of this type. They are identified by the rule name prefix *syntax*.

QA/Slam rules that locate syntax exceptions are redundant. A syntactic analysis is carried out, both in the analysis phase when a Slam II program is loaded into E/Slam, and in the update phase when a program modification is made using the template edit feature. Nevertheless, a small number of rules are included in the rule-base to demonstrate that QA/Slam is extensible in this regard. QA/Slam and E/Slam are separate run time modules, thus QA/Slam could

conceivably be used as a component of another tool that allows parameter field updates yet does not carry out its own syntax checks at the time of the update. As long as the assertion holds that the program is syntactically correct when loaded into the environment and the environment allows updates on a parameter field basis only, then one can conclude that if a syntactic error exists in the program it is a result of the new value placed in the parameter field.

Semantic exceptions are classified further according to their severity. Two levels of severity exist: *Warning* and *Error*. Warning exceptions reflect situations where the Slam II language is used in a syntactically correct way, yet possibly in a semantically incorrect way. Due to the nature of the inconsistency QA/Slam cannot conclude if it is an actual error. This situation arises either when the inconsistency is minor —and thus may be intentional— or because there is insufficient information in the Slam II program to draw a definite conclusion about the programming practice. QA/Slam makes the user aware of programming practices that could be indicative of a program design problem. It is the users responsibility to investigate the programming practice that caused the warning, to ensure that it was actually intended.

An error exception identifies a programming practice that is conclusively incorrect. An error can be semantic or syntactic. A semantic error, unlike a semantic warning, identifies a coding practice that will either cause a run time error or cause incorrect simulation results to be generated, unless the inconsistency is resolved.

Semantic exceptions are partitioned into range exceptions and mode exceptions. Range exceptions and mode exceptions are defined in Section 8.1. A

range exception can either be a warning or an error. Knowledge-base rules that identify range warnings have *warnRange* as their rule name prefix. Rules that identify range errors have *errorRange* as their prefix. Similarly rules identify mode warnings have *warnMode* as a prefix and rules that identify mode errors have *errorMode* as a prefix.

Another way in which exceptions can be classified is according to how the error is manifested within the program or the scope of analysis required to identify the inconsistency. As shown in **Figure 8.13**, this classification gives rise to two types of analysis: single field checks and multi-field checks.

Type of Analysis	Scope of Analysis	Value is Inconsistent With
Single field checks	A single field within a statement	Supporting system
Multi-field checks	Multiple fields within a single statement	Other field within same statement
	Multiple fields not all within a single statement	Functional element outside of statement

Figure 8.13 Different scopes of field level inconsistency

Single field checks involve investigating a single parameter field within the Slam II program. In this situation a specification is invalid if it is inconsistent with respect to the Slam II supporting system. Syntax errors and some types of range errors fall into this category. An example of a syntax error would be a field that has a variable specified as its value when it is only allowed to take on a constant value. An example of a range error would be the specification of a negative value in the queue capacity (QC) field of a QUEUE statement.

Multi-field checks require more than one portion of the program to be investigated in order to identify an inconsistency in program specification. This investigation can remain local to the statement, or non-local to the statement. Investigations local to a statement involve comparing the contents of multiple parameter fields within the single statement. Non-local investigation involves comparing the contents of a parameter field in one statement with the parameter field value of another statement, the existence of another statement or program element.

8.5.3 Diagnostics File

The diagnostics file holds parameterized diagnostic message text that is used to document inconsistencies found in the Slam II program. When an inconsistency is discovered in the program, details such as the name of the rule that fired, the severity of the exception, statement number, statement name, field name, field offset, and field value are recorded in the QA report file. In addition to this, a diagnosis of the problem is given together with one or more suggested fixes for the problem.

The diagnostics file distinguishes exceptions according to the number of Slam II statement parameter fields involved in the inconsistency. To document the occurrence of an exception, the consequent of each diagnostic rule contains a predicate type_*x*_exception() where *x* is equal to 0, 1, 2 or *n*. The value *x* is a count of the number of statement fields involved in the inconsistency. Type 0 exceptions represent inconsistencies at the statement level instead of at the statement field level, thus the number of parameter fields involved in the inconsistency is zero. A type *n* exception is any inconsistency that involves greater than two parameter fields.

Documentation Levels

There are three levels of documentation held in the diagnostics file, (1) summary information, (2) preamble information, (3) detailed diagnostic and suggest fix information.

Summary Information

The summary information includes a condensed description of the elements involved in the inconsistency, their location in the program, the severity of the exception, and the name of the rule that fired. This section is parameterized so that when it comes time to document an inconsistency, the parameters are filled in with the appropriate information.

Preamble Information

The preamble explains, in sentence form, the type of inconsistency discovered. It presents the summary section's information in paragraph form for easier understanding. There are three different exception types: syntax exception, range exception, and mode exception.

Like the summary section, the preamble section is parametric in nature. This facilitates a more precise description of the inconsistency, since the contributing factors can be explicitly named within the paragraph.

Detailed Information of Diagnosis and Suggestions

The detailed diagnostic information provides a detailed description of the inconsistency and is accompanied with suggestions on how to remedy the situation. Each QA rule in the rule-base has a corresponding detailed diagnostic entry and suggested fix entry in the diagnostics file to describe the inconsistency in detail. The text in this section is also parametric in order to facilitate a more precise explanation.

Documentation Steps

As shown in **Figure 8.14**, when a diagnostic rule fires, it initiates documentation by branching to the appropriate exception type. At level 1 the summary information regarding the inconsistency is generated and placed in the QA report. If it is a type *n* exception no documentation is generated. This can be interpreted as a null action at level 1.

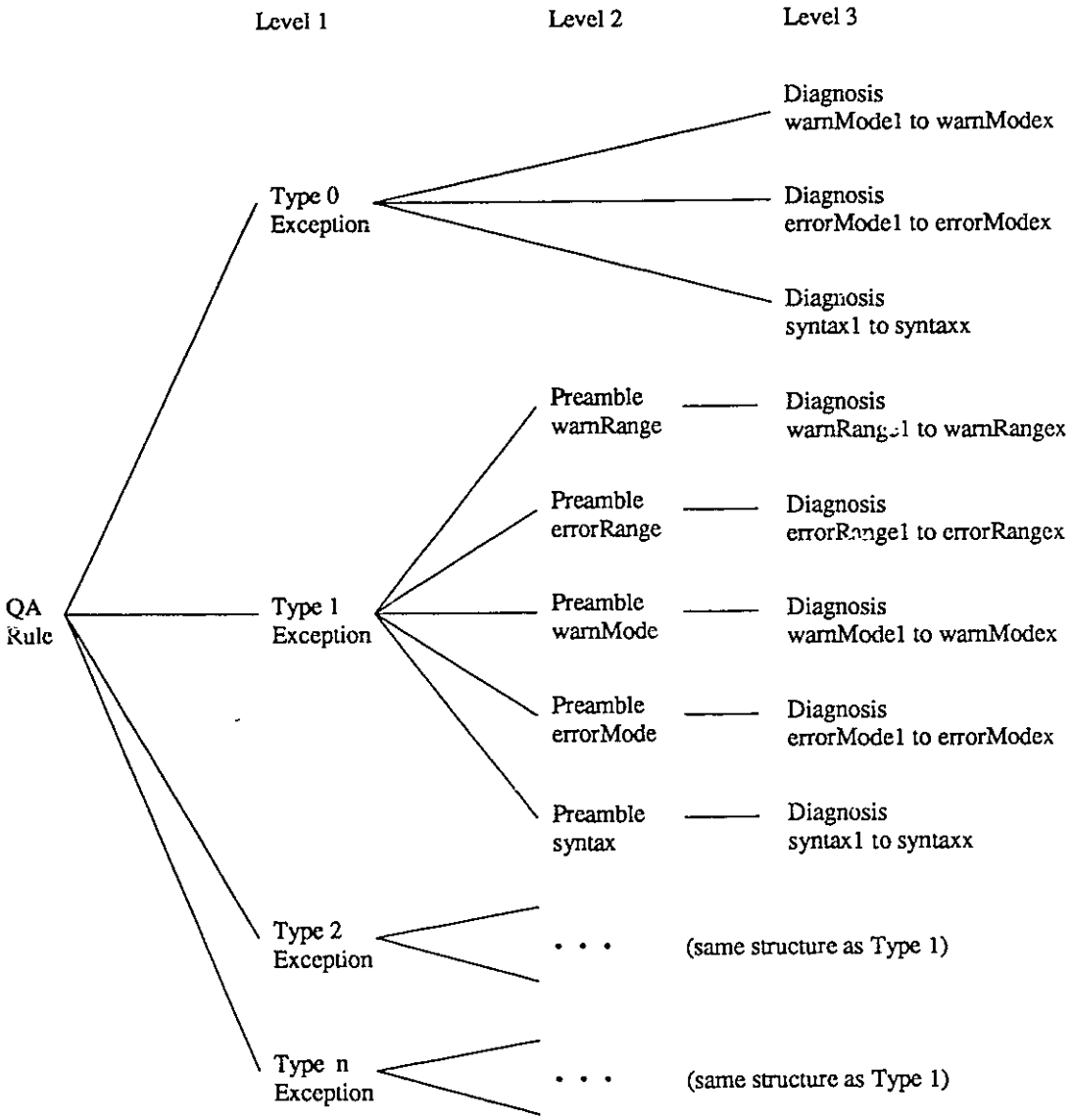


Figure 8.14 Tree structure of QA/Slam documentation steps

Upon completion of level 1 documentation, the system proceeds to level 2 where the preamble description of the inconsistency is generated and placed in the QA report. What is said in the preamble section depends on the rule type that fired. A different preamble exists for each rule type. For example, if the rule warnRange1 fires then the preamble for warnRange is chosen.

Following the preamble stage of documentation, the system proceeds to level 3 where the detailed diagnostic messages and suggested fix messages are generated and placed in the QA report. These messages are indexed by rule name and a one-to-one correspondence exists between QA rules and level 3 message.

8.5.4 QA/Slam Inference Engine

The QA/Slam inference engine is the system that drives the consistency checking operations. Running on top of Quintus Prolog, it applies the rules in the rule-base to the Slam II program facts in the fact base and working memory, to identify inconsistencies in program specification. When inconsistencies are found, it initiates the documentation of the inconsistency, instantiating the parameterized diagnostics message text to the program elements that contributed to the inconsistency. Through the Prolog supporting system, the QA/Slam inference engine also backtracks where necessary to locate all occurrences of an inconsistency.

The QA/Slam inference engine is made up of six components: (1) an inferencing mechanism, (2) initialization and fact building functions, (3) extraction functions, (4) comparison and test functions, (5) graph connectivity functions, and (6) miscellaneous functions. Each is distinguished according to the kind of service it provides and is discussed in the following sections.

```
go :-
      call(rule ID# LHS ==> RHS),
      try(LHS,RHS),
      fail.

try(LHS,RHS) :-
      match_premise(LHS),
      process(RHS),
      !.

match_premise([]).
match_premise([N#Premise | Rest]) :-
      call(Premise),
      match_premise(Rest).

process([]).
process([X|Y]) :-
      call(X),
      process(Y).
```

Figure 8.15 QA/Slam inferencing mechanism

Inferencing Mechanism

The inferencing mechanism is the QA/Slam kernel. It sits on top of the Quintus Prolog interpreter and is the center of all activity. The inferencing mechanism is responsible for selecting QA rules, evaluating their premises, and when all premises are true, it is responsible for performing the rule action.

The top level of the inferencing mechanism is represented by the predicate named "go" in **Figure 8.15**. The inferencing mechanism steps through the rule-base sequentially with the predicate "call(rule ID# LHA ==> RHS)". When a rule is located, LHS instantiates to the rule premise and RHS instantiates to the rule consequent.

When a rule is selected for analysis the inference engine steps through the rule premises one by one. If the current premise evaluates to true, then the inference engine follows through to the next premise in the list. This is handled by the match_premise() predicate in **Figure 8.15**. When a premise evaluates to false, backtracking occurs for that premise and an alternate instantiation is sought for the predicate. If an instantiation is found, then the inferencing mechanism moves forward once again seeking an instantiation for the next premise.

When all premises evaluate to true (i.e., the match_premise() predicate succeeds), the rule fires and the predicates listed in the consequent of the rule are evaluated. This step is handled by the process() predicate. Any instantiations from the premises that would supply useful information to the diagnostics message for the inconsistency are passed in the consequent of the rule.

After the consequent is evaluated, the inference engine backtracks on the rule premises, to find other instantiations that cause the rule antecedent to evaluate to true. This is initiated by the *fail* predicate. If other instantiations are found then once again evaluation follows through to the rule consequent. If there is no way for the antecedent to evaluate to true given the set of facts present in working memory, then the rule fails. At this point the inferencing mechanism moves on to the next rule in the rule-base. The QA/Slam inferencing mechanism evaluates the rules in a linear fashion, according to the order in which they are listed in the knowledge-base file. Because the number of rules in the knowledge-base is not large, and because the input fact base is generally not that large, a linear rule search is a reasonable strategy.

Initialization and Fact Building Functions

The initialization and fact building functions are called as one of the first steps in QA/Slam's execution. These functions are responsible for setting up the operating environment. This involves loading the fact-base, rule-base, diagnostics file, and preparing the QA report file to accept messages. **Figure 8.16** and **8.17** show the predicates that perform this role.

Once the fact base is loaded, it is analyzed to identify relationships that will be frequently drawn upon as the knowledge-base rules are evaluated. The discovered relationships are expressed as separate facts in order to make the information quickly and easily accessible by the rules.

Information regarding how file numbers are associated with QUEUES, RESOURCE blocks and GATE blocks is an example of information frequently referenced by QA rules. To determine the relationships it is necessary to search all statement facts related to GATE, RESOURCE, and QUEUE statements to see what file numbers are associated with them. Once this is complete, additional facts are generated to express the discovered relationships explicitly.

The init_file_declaration_table() predicate extracts information from the Slam II fact base regarding what file numbers are assigned to what resources, gates, and queues. This information is entered into working memory as a predicate

$$declaration(IFL, LBL, Ftype)$$

where IFL represents the file number, LBL indicates a RESOURCE label or GATE label associated with the file number, and Ftype indicates whether the file number is associated with a resource, gate or queue.

```
/*
 | The following predicate inspects all gate, resource and
 | queue statement facts to determine which file numbers
 | are being used by each, and generates a fact describing
 | the associations. Each fact has the following format:
 |     fact(declaration(Ifl,Lbl,Obj)
 | where:
 |     Ifl : file number
 |     Lbl : Gate label or Resource label
 |     Obj : gate, resource or queue
 */
init_file_declaration_table :-
    gate_declarations,              /*collect gate facts*/
    resource_declarations,          /*    "    resrc  "  */
    queue_declarations.             /*    "    queue  "  */
/*
 |  Determine all gate/file associations
 */
gate_declarations    :-
   cont('GLBL','GATE',Ln,_,_,GLBLvalue),
   cont('IFL','GATE',Ln,_,_,IFLvalue),
   field_value(IFLvalue,Ifl),
   field_value(GLBLvalue,Glbl),
   not(factbase:fact(declaration(Ifl,Glbl,gate))),
   assertz(factbase:fact(declaration(Ifl,Glbl,gate))),
   fail.                            /* search for more */
gate_declarations.

/*
 |  Determine all resource/file associations
 */
resource_declarations    :-
   cont('RLBL','RESOURCE',Ln,_,_,RLBLvalue),
   cont('IFL','RESOURCE',Ln,_,_,IFLvalue),
   field_value(IFLvalue,Ifl),
   field_value(RLBLvalue,Rlbl),
   not(factbase:fact(declaration(Ifl,Rlbl,resource))),
   assertz(factbase:fact(declaration(Ifl,Rlbl,resource))),
   fail.
resource_declarations.
```

Figure 8.16 Extract of QA/Slam showing initialization and fact building functions

```
/*
 |  Determine all queue/file associations
 */

queue_declarations :-
    cont('IFL','QUEUE',Ln,Slbl,_,IFLvalue),
    queue_declarations_aux(IFLvalue,Slbl),
    fail.
queue_declarations.
/*
 |  The following predicate handles an entry of the
 |  form ATRIB(I)=J,K in the IFL field
 */
queue_declarations_aux(Pval,Lbl) :-
    file_range(Pval,I,J,K),
    queue_range_aux(J,K,Lbl),!.
/*
 |  The following predicate handles a numeric constant
 |  in the IFL field
 */
queue_declarations_aux(PVal,Lbl) :-
    field_value(PVal,Val),
    not(factbase:fact(declaration(Val,Lbl,queue))),
    assertz(factbase:fact(declaration(Val,Lbl,queue))),
    !.
queue_declarations_aux(_,_).  /* taken when fact already in KB */
/*
 |  Extract every file number in the range J to K and
 |  assert a fact for eac one
 */

queue_range_aux(J,K,Lbl) :- J > K.
queue_range_aux(J,K,Lbl) :-
   J =< K,
   J1 is J + 1,
   queue_range_aux(J1,K,Lbl),
   queue_range_aux1(J,Lbl).
queue_range_aux1(J,Lbl) :-
   not(factbase:fact(declaration(J,Lbl,queue))),
   assertz(factbase:fact(declaration(J,Lbl,queue))).

queue_range_aux1(_,_).        /* taken when fact already in KB. */
```

Figure 8.17 Extract of QA/Slam showing initialization and fact building functions

Extraction Functions

The extractor functions shown in **Figure 8.18**, provide an interface between the rules in the rule-base and the facts in the fact base. They allow one to access statement field information by field name, statement name, statement number, statement label, or any combination of these criteria.

The advantage of this is that it shields one from the list processing required to extract a parameter field value from a statement fact. Such extraction involves a certain degree of list processing since the statement facts store Slam II statement information in the Prolog lists and sublists. Supplying extraction functions to handle this list processing allows one to concentrate on the relevant data itself instead of concentrating on how to locate this relevant data. The extractor functions make information about Slam II statements, available for subsequent processing by QA/Slam's comparison and test functions, connectivity functions and miscellaneous functions.

Two predicates fall into this category, the *content* predicate cont() and get-parameter() predicate. The cont() predicate is used to instantiate with statement() facts in the fact base, according to field id, statement id, line number, statement label, field offset, or any combination of these. Once an instantiation is found, the getparameter() predicate extracts information on the parameter field requested and instantiates this information with the Pl variable.

```
cont(Fid,Sid,Ln,Slbl,Ofst,Pl) :-
     factbase:statement(Slbl,Ln,Sid,Plist),
     getparameter(Fid,Ofst,Plist,Pl).
getparameter(Fid,Off,[[Fid,Off,Val] | Rest],[Fid,Off,Val]).
getparameter(Fid,Ofst,[H|T],Pl) :- getparameter(Fid,Ofst,T,Pl).
```

Figure 8.18 Extract of QA/Slam showing the cont() predicate and getparameter() predicate

Comparison and Test Functions

The comparison and test component offers a set of functions for comparing values and testing values for the presence of certain properties. The test functions can be used to query whether an argument is a resource label, gate label, resource file number, gate file number, constant, variable, default value, or conditional expression.

The comparison functions allow one to compare the content of a Slam II parameter fields with each other and with constant values. All comparisons involving Slam II statement parameter fields expect field values to be in Pvp format. This is advantageous since the fact base has all statement parameter field values expressed in this format. This simplifies comparisons because it treats a parameter field's name, offset, default flag, and field value as an atomic unit, thus eliminating level of list processing.

When a comparison is made involving a constant operand, this operand need not be expressed in Pvp format. A set of functions called immediate test functions is provided which allow a constant value to be expressed as an operand.

```
test([_,_,[_,inf]],'>',[_,_,[_,Rhs|T2]]) :-
     Rhs \=  'inf'.
test([_,_,[_,Lhs|T1]],'>',[_,_,[_,Rhs|T2]]) :-
     is_constant([_,_,[_,Lhs|T1]]),
     is_constant([_,_,[_,Rhs|T2]]),
     atom_to_num(Lhs,L),
     atom_to_num(Rhs,R),
     L > R.

itest([_,_,[_,inf]],'>',Ival) :-
     Ival \= inf.
itest([_,_,[_,Lhs|T]],'>',Ival) :-
     !,
     is_constant([_,_,[_,Lhs|T]]),
     constant(Ival),
     atom_to_num(Lhs,L),
     atom_to_num(Ival,I),
     L > I.
```

Figure 8.19 Sample test() and itest() predicates

A subset of QA/Slam's test functions is shown in **Figure 8.19**. The test() predicates expect both operands to be statement field values expressed in Pvp format. The testi() predicate expects a statement field value in Pvp format as one of its operands, and a constant value as its second operand.

Graph Connectivity Functions

The graph connectivity functions enable one to reference the topology of a Slam II program's network graph. These functions use the connect() predicates in the fact base, to determine the graphical predecessor and successor of a network node. This makes it possible for one to define rules that check for inconsistencies

related to program topology. **Figure 8.20** and **8.21** show the QA/Slam graph connectivity functions.

Miscellaneous Functions

The miscellaneous functions act as supporting functions to the other higher level functions in QA/Slam. They generally operate at the Prolog list structure level, allowing one to maximize use of Prolog's language constructs. Functions for appending elements to lists, determining the length of a list, enumerating elements of a list and determining how many characters exist in an atom belong to this category. Functions that output information to the standard output device also belong here.

8.6 Prolog System

Quintus Prolog Version 3.1.1 is the system upon which QA/Slam operates. Prolog was chosen as the implementation language because of its built-in unification feature, backtracking feature and inferencing mechanism. Most of QA/Slam's consistency checking activity involves comparing the content of Slam II statement fields. The unification feature is useful here because it reduces complexity by providing a means for easy assignment of Slam II statement parameter field values to variables for manipulation.

In a Slam II program, it is possible for an inconsistency identified by a QA rule to reoccur at multiple locations in the program. Consequently, once the rule fires, it is necessary to backtrack on the premises of the rule to determine if

```
  /*
   | predecessors(+Ln,+Ltype,-Olist)
   | Returns a reference to those statements that preceded
   | the statement at line number <Ln> in the Slam II network
   | model.
   |
   | Instantiates <Olist> to a list of [link-type,line-number]
   | pairs where line-number is the predecessor statement's
   | line number and <link-type> identifies the type of
   | link that connects the statements. <Ltype> specifies what
   | link types are to be inserted into list <Olist>.
   |
   | <Ltype> can take on the value 'normal' for normal link,
   | 'balk' for balk link, or 'all' for both types of links.
   */
predecessors(Ln,Ltype,Olist) :-
     findall([C,Y],factbase:connect(Y,C,Ln),Slist),
     extract_type(Ltype,Slist,Olist).
  /*
   | predecessor_is_x(+Ltype,+Ln,+Sid,-Predln)
   | <Predln> is instantiated to the line number
   | of a predecessor node, having statement type <Sid>,
   | that routes directly to the statement at line number
   | <Ln> via a link of type <Ltype>.
   */

predecessor_is_x(Ltype,Ln,Sid,Predln) :-
     predecessors(Ln,Ltype,Predlist),
     sid_match(Sid,Predlist,Predln).
  /*
   | predecessors_are_x(+Ltype,+Ln,+Sid,-Predlnlist)
   | Analogous to predecessor_is_x/4 except
   | <Predlnlist> is instantiated to all predecessors
   | that match the criteria.
   */
predecessors_are_x(Ltype,Ln,Sid,Predlnlist) :-
     predecessors(Ln,Ltype,Predlist),
     sid_match_all(Sid,Predlist,Predlnlist).
```

Figure 8.20 Graph connectivity predicates in Prolog

```
  /*
   | successors(+Ln,+Stype,-Olist)
   | Analogous to predecessors/3 except deals with
   | successor nodes to <Ln>.
   */
successors(Ln,Stype,Olist) :-
     findall([C,Y],factbase:connect(Ln,C,Y),Slist),
     extract_type(Stype,Slist,Olist).

  /*
   | successor_is_x(+Type,+Ln,+Sid,-Sln)
   | Analogous to predecessor_is_x/4 except deals with
   | successor nodes to <Ln>.
   */
successor_is_x(Type,Ln,Sid,Sln) :-
     successors(Ln,Type,Slist),
     sid_match(Sid,Slist,Sln).

  /*
   | successors_are_x(+Type,+Ln,+Sid,-Slnlist)
   | Analogous to predecessors_are_x/4.
   */
successors_are_x(Type,Ln,Sid,Slnlist) :-
     successors(Ln,Type,Slist),
     sid_match_all(Sid,Slist,Slnlist).
```

Figure 8.21 Graph connectivity predicates in Prolog

any other occurrences of that inconsistency exist. Prolog's built-in backtracking feature is used to elegantly facilitate this requirement.

8.7 QA Report

Every time a program inconsistency is discovered, an entry is made in the QA report file. **Figure 8.22** shows the entry format for a type 2 exception (i.e., an inconsistency involving two statement fields). Aside from the summary section, type 0, 1, 2, and *n* exceptions all have the same format in which a message area comprised of a preamble section and a detailed diagnosis section, is followed by a suggestions section.

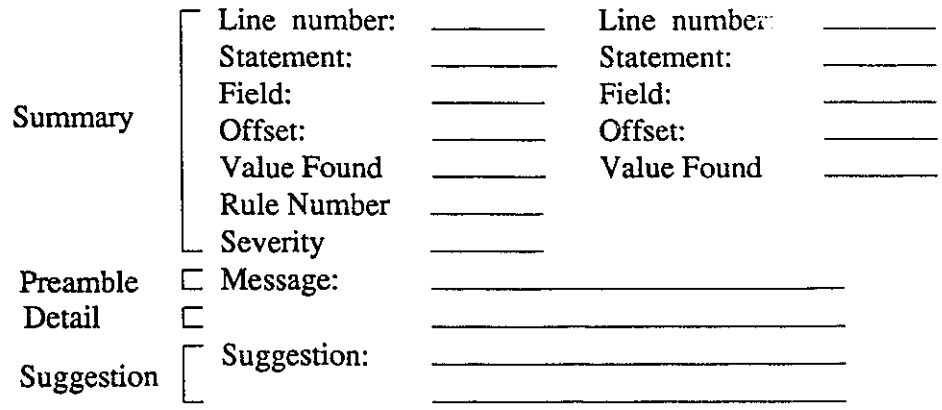


Figure 8.22 QA report structure

The summary section differs for each exception type. This is because each exception type has associated with it, a different degree of information that must be communicated to the user. The summary section for a type 2 exception includes a description of the location of each statement field involved in the inconsistency, the inconsistent values, the name of the rule that identified the inconsistency, and the severity of the inconsistency. The location of the inconsistency is identified by supplying the line number, statement name, field name, and field offset for each offending statement.

Because type 2 exceptions identify inconsistencies between two statement fields, the summary section supplies information on the location of both fields involved in the inconsistency. Type 1 exceptions, on the other hand, have only one column of location information, since they need identify only one statement parameter field entry. Type 0 exceptions and type n exceptions have no summary section at all since the nature of these exception types does not warrant it.

Figure 8.23, Figure 8.24 and **Figure 8.25** show segments of a QA report generated by QA/Slam for the Slam II program in **Figure 2.7**. The rules that fired are errorMode7, warnMode4, and errorRange5. This report would be displayed in a text window automatically by E/Slam for the analyst to browse.

```
Line number: 13          Line number: 77
Statement:   GATE        Statement:   QUEUE
Field:       IFL         Field:       IFL
Offset:      4           Offset:      1
Value Found: 3           Value Found: 3
Rule Number: errorMode7
Severity:    ERROR
Message:
    The specified value of 3
    in the file-identification field (IFL) of the GATE
    statement on line number 13 is semantically incorrect
    with respect to the value 3
    in the file-identification field IFL of the
    QUEUE statement on line number 77.
    It is not permissible to associate a file with both a
    GATE and a QUEUE simultaneously in the network model.
SUGGESTION:
    Remove the duplication of file number 3
    from one of the above mentioned statements.
```

Figure 8.23 Segment of QA report generated by QA/Slam for the Slam II program in **Figure 2.7**. Output is shown for rule errorMode7

```
Line number: 22          Line number: 22
Statement:   CREATE      Statement:   CREATE
Field:       TBC         Field:       MC
Offset:      1           Offset:      4
Value Found: UNFRM(240)  Value Found: 1
Rule Number: warnMode4
Severity:    WARNING
Message:
    The specified value of UNFRM(240)
    in the time-between-entity-creations field (TBC)
    of the CREATE statement on line number 22 is out of
    context with respect to the value 3
    in the maximum-creations field (MC)
    of the CREATE statement on line number 22
    and thus may be indicative of a specification error.

    The MC field indicates only 1 entity is to be created by
    this node, yet an explicit value has been assigned to the
    TBC field, which implies that more than one entity is
    expected to be released.
SUGGESTION:
    If only one entity is to be released then remove the value
    from the TBC field and leave it defaulted.
    OR
    If more than one entity is to be released change the value
    in the MC field.
```

Figure 8.24 Segment of QA report generated by QA/Slam for the Slam II program in **Figure 2.7**. Output is shown for rule warnMode4

```
Line number: 82          Line number: 10
Statement:   ASSIGN      Statement:   LIMITS
Field:       VAR         Field:       MATR
Offset:      1           Offset:      2
Value Found: ATR(22)     Value Found: 5
Rule Number: errorRange5
Severity:    ERROR
Message:
    The specified value ATR(22) in the variable field (VAR)
    of the ASSIGN statement on line number 82
    is out of range with respect to the specified value
    5 in the MATR field of the LIMITS statement on line
    number 10.
    The index 22 specified for ATRIB() exceeds the
    maximum allowable index value of 5 specified in
    the LIMITS statement.
SUGGESTION:
    Ensure that the argument to ATR(22) is the desired argument.
    AND/OR
    Increase the value in the MATR field of the LIMITS statement
    to a value greater than or equal to 22.
```

Figure 8.25 Segment of QA report generated by QA/Slam for the Slam II program in **Figure 2.7**. Output is shown for rule errorRange5

9 Conclusions and Future Work

Throughout this thesis we have demonstrated solutions to some of the problems facing us in relation to our growing inventory of simulation programs. We showed how a program understanding system could be used to exceed the capabilities of traditional forms of software documentation. We showed how a program understanding system can be used to expand the size of audience that can interact with the existing base of simulation programs, in order to specify and run new simulation experiments. We also showed how a program understanding system can support program changes while ensuring program quality is not compromised. These goals were achieved in the design of E/Slam by providing four features:

1. a program understanding system that elucidates information through two types of documentation templates: statement templates and program templates.
2. knowledge-based editing,
3. knowledge-based reliable editing (built in quality assurance),
4. program synthesis.

Program Understanding System with Documentation Templates

By designing E/Slam as a program understanding system that elucidates program information through documentation templates, we are able to present the acquired knowledge about a Slam II program, to the user in a way that is easier to understand than the program source code. This is because E/Slam's program understanding ability involves having knowledge about implicit Slam II program information, explicit Slam II program information, and knowledge about

the Slam II supporting system. The documentation templates act as a venue for the program understanding system to elucidate program information in logical way, unaccompanied by syntactic details of the implementation language. As an aside, in order to systematize the implicit and explicit information held in Slam II statements it was necessary to produce finite state acceptors for each of the 43 language statements.

By designing E/Slam to have knowledge about the functional elements of conventional simulation programs, we are able to elucidate program information from two orthogonal perspectives: the statement perspective and the program perspective. The statement perspective is achieved through statement templates that elucidate relevant program information on a per-statement basis where information on all occurrences of a given type of statement is brought together under one template. The program perspective is achieved through program templates that obtain their information by taking a vertical slice of the program, where each vertical slice corresponds with one of four functional categories: initialization, input scheduling, behavior processing and termination conditions.

Program understanding ability and elucidation via statement templates and program templates enables E/Slam to fill the gap left by insufficient documentation, in addition to expanding the size of audience that can interact with the simulation program.

Knowledge-Based Editing

By supporting knowledge-based editing E/Slam is able to ensure program updates are syntactically correct. Pending updates are subjected to a detailed

syntactic analysis, and the update is accepted only if it passes the syntax analysis phase.

Editable documentation templates are an important feature because they provide a venue for the analyst to specify new experiments. Furthermore, because the updates are made through the templates, the analyst is not required to interact directly with the source code. E/Slam handles the job of manifesting the specified changes in the appropriate parts of the program. The advantage of this is that it enables one who does not have an intermediate or advanced understanding of the implementation language, to specify new simulation experiments.

Knowledge-Based Reliable Editing

The author has stressed that if a program understanding system offers an environment for editing programs, it should provide an environment for reliable editing. Especially if the program understanding system has been designed to enable analysts who are not experts in the implementation language to make program changes. This can be realized by designing an embedded rule-based system consisting of a knowledge-base that identifies what constitutes an inconsistency in program specification. The knowledge-base rules can be applied to a fact base of information about the program under study. If inconsistencies are discovered, the analyst could be notified of the inconsistencies and given advice on how to alleviate them.

We are able to instill a higher degree of confidence of program correctness in the mind of the analyst specifying new Slam II simulation experiments, by offering knowledge-based reliable editing. Through QA/Slam and its knowledge

about possible sources of inconsistency in program specification, E/Slam is able to study a simulation program to ensure changes made by the analyst have not caused inconsistencies to come about. Through knowledge-based reliable editing, E/Slam is able to address the cited concerns regarding the proper validation of program changes.

Program Synthesis

The ability to keep tabs on sources of information from which documentation template entries are generated, allows E/Slam to offer an environment for program synthesis. This, combined with documentation templates that provide a venue for updating experiments, allows changes to be made to the simulation program source code without interacting directly with the program source code. Furthermore, E/Slam's ability to keep the user interface consistent with updates made to the internal model of the Slam II program, keeps the user informed about the current state of the program at all times. Once the analyst has completed making a set of program updates, E/Slam is able to generate a snap shot of the current state of the internal model as Slam II source code. The source code is then ready to be input directly to a Slam II compiler to produce an executable version of the updated program.

Future Work

The following items are suggestions about possible future work that could be undertaken.

- The current design of E/Slam includes statement templates (for Slam II network, control and M.H.E. statements), program templates, template editor, program generator, quality assurance component and internal model structure. The current implementation of E/Slam includes statement templates, template editor, program generator, quality assurance component (QA/Slam) and internal model structure. Possible future work could be implementation of the M.H.E. (material handling extension) templates and program templates.
- E/Slam currently supports program updates that involve changing the value specified in parameter fields of existing program statements. This could be expanded upon. E/Slam could be enhanced to insert entire statements into the program based on user specifications. This would give the user more control and flexibility in the kinds of program updates that could be made. The ability to add statements to the program would give the analyst the ability to make a wider variety of changes to pre-run conditions and pre-study conditions. It would also enable the user to increase the number of simulation runs that could be specified in a Slam II program. The user interface for such an enhancement should be designed in such a way that the user is not required to know the syntax of the statements added. The analyst should be able to specify new parameter values and add simulation runs through the template interface.

- The concepts generated in the design and development of E/Slam are general ones, and can thus be applied to programming languages other than Slam II. Possible future work could involve applying the principles outlined in this thesis, to other simulation languages like GPSS, or general purpose languages like C++.

Appendix A
E/Slam Program Templates

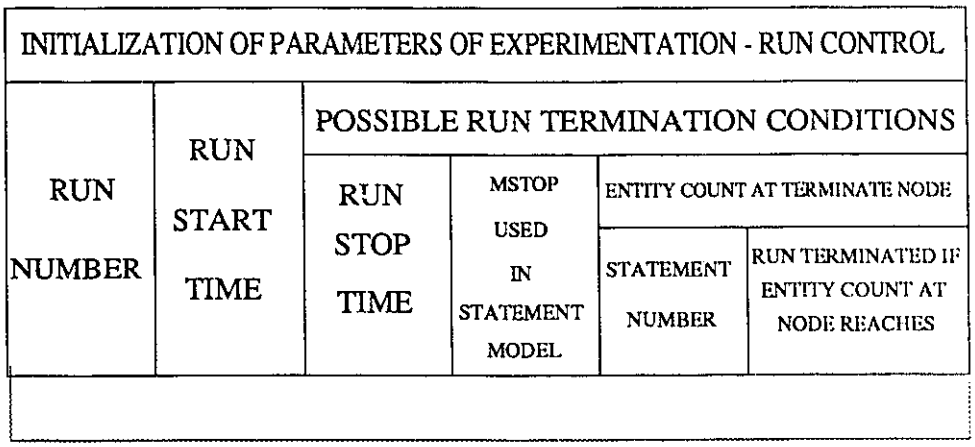


Figure A.1 E/Slam's "Initialization of Parameters of Experimentation —Run Control" template

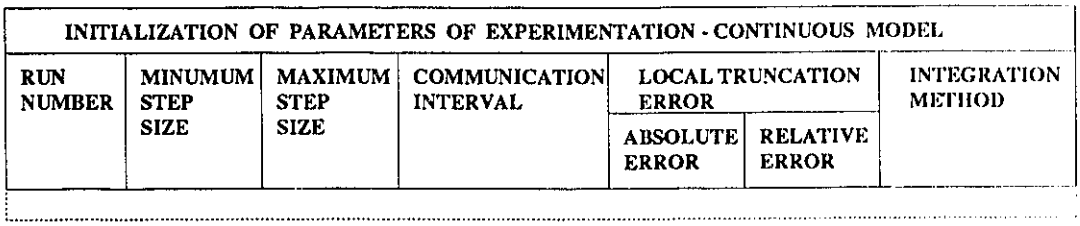


Figure A.2 E/Slam's "initialization of parameters of experimentation —continuous model" template

RANDOM NUMBER STREAM INITIALIZATION			
RUN NUMBER	STREAM NUMBER	SEED VALUE	REINITIALIZE STREAM BETWEEN RUNS

Figure A.3 E/Slam's "random number stream initialization" template

IDENTIFIER INITIALIZATION				
RUN NUMBER	IDENTIFIER NAME	ALIAS NAME	CLASSIFICATION OF IDENTIFIER	INITIAL VALUE

Figure A.4 E/Slam's "identifier initialization" template

FILE INITIALIZATION						
RUN NUMBER	FILE NUMBER	ALIAS NAME	FILE TYPE	FILE CONTENT AT START OF RUN	ENTITY ATTRIBUTES	
					NAME	VALUE

Figure A.5 E/Slam's "file initialization" template

GATE		
NAME	NUMBER	INITIAL STATUS

Figure A.6 E/Slams "gate status" template

CREATE,TBC,TF,MA,MC,M;						
STATEMENT		TIME			MAXIMUM	
NUMBER	LABEL	BETWEEN ENTITY CREATIONS	OF FIRST ENTITY CREATION	OF ENTITY CREATION SAVED IN	ENTITY CREATIONS	NUMBER OF INITIATED ACTIVITIES PER RELEASE

Figure A.7 E/Slam's "entity arrival scheduling" template. This is an alias for the CREATE template

TABLE LOOKUP FUNCTION					
INDEPENDENT VARIABLE		DEPENDENT VARIABLE	RESULTS RETURNED TO		
VALUE	REFERENCE ROW NUMBER	REFERENCE ROW NUMBER	STATEMENT NUMBER	STATEMENT TYPE	FIELD

Figure A.8 E/Slam's "tabular input" template

TABLES AND PLOTS GENERATED					
RUN NUMBER	STATISTICS COLLECTED ON	DESCRIPTION	OUTPUT DISPLAYED AS	STATISTICS COLLECTED AT	
				STATEMENT	LINE NUMBER

Figure A.9 E/Slam's "tables and plots" template

SIMULATION OUTPUT			
RUN NUMBER	STATEMENT LISTING	ECHO REPORT	SUMMARY REPORT

Figure A.10 E/Slam's "general simulation output" template

PROGRAM TRACE							
RUN NUMBER	ERROR REPORT	SUMMARY REPORT	CONTENT OF FILES	VALUE OF CONTINUOUS VARIABLES	OF SYSTEM STATE	TRACE	
						NODES TO BE TRACED	PRINT VALUE OF

Figure A.11 E/Slam's "program trace data" template

IDENTIFIER USAGE					
IDENTIFIER	**ALIAS**	**CLASSIFICATION**	**DESCRIPTION OF USAGE**	**USED AT**	
				STATEMENT	**LINE NUMBER**

Figure A.12 E/Slam's "identifier usage" template

FILE USAGE				
FILE NUMBER	ALIAS NAME	FILE TYPE	USED AT	
			STATEMENT TYPE	STATEMENT NUMBER

Figure A.13 E/Slam's "file usage" template

ELEMENTS SHARED AMONG SUBNETWORKS					
SHARED ELEMENT			SUBNETWORK STATEMENT RANGE		ELEMENT DEFINITELY USED
ELEMENT TYPE	ELEMENT		FROM STATEMENT NUMBER	TO STATEMENT NUMBER	
	NAME	NUMBER			

Figure A.14 E/Slam's "usages across subnetworks" template

STATEMENT NUMBER	GATE					
	NUMBER	NAME	INITIAL STATUS	WAITING QUEUE FILE NUMBER	USED AT	
					LINE NUMBER	STATEMENT TYPE

Figure A.15 E/Slam's "gate usage" template. This is an alias for the GATE template

STATEMENT NUMBER	RESOURCE					
	NUMBER	NAME	UNITS AVAILABLE	WAITING QUEUE FILE NUMBER	USED AT	
					LINE NUMBER	STATEMENT TYPE

Figure A.16 E/Slam's "resource usage" template. This is an alias for the RESOURCE template

References

Aho A.V., Sethi R., Ullman J.D. (1986), "Compiler Principles, Techniques, and Tools," Addison-Wesley, Reading, Massachusetts, NY.

Balci O. (1992), "Guidelines for Successful Simulation Studies," In: Proceedings of the Advances in Simulation '92 Symposium, A. R. Kayla and T. I. Ören, (eds.), July 6–7, 1992, Estanbul, Turkey, Bogazici University, Estanbul, Turkey, pp. 2–16.

Birta L.G., Abou-Rabia O., Ören T.I., King D., Wendt R. (1991), "Reverse Engineering in The Simulation Lifecycle," SAMS, Vol. 9, pp. 69–84.

Böcker, H.D. (1991), "The Role of Visual Representations in Understanding Software," In: Artificial Intelligence and Software Engineering, D. Partridge (ed.), Ablex Publishing Corporation, Norwood, NJ.

Golshani, F. (1990), "Rule-Based Expert Systems," In: Knowledge Engineering Fundamentals Vol. 1, McGraw-Hill, New York, NY.

King D., Ören T.I., Hitz M. (1992), "Automatic Generation of Natural Language Documentation for Slam II Programs," In: IMACS Transactions on Scientific Computing '91, E.N. Houstis and J.R. Rice (eds.), North-Holland, Amsterdam.

Myers G.J. (1979), "The Art of Software Testing," John Wiley and Sons, New York, NY.

Nielson N.R. (1991), "Application of Artificial Intelligence Techniques to Simulation," In: Knowledge-Based Simulation, P.A. Fishwick and R.B. Modjeski (eds.), Springer-Verlag, Berlin, New York, pp. 1–9.

Ning J.Q. (1992), "Panel on: Program Understanding—Does it Offer Hope for Aging Software?," In: The Seventh Knowledge-based Software Engineering Conference, McLean, Virginia, Sept 20–23, 1992.

O'Reilly J.J., Lilegdon W.R. (1987), "SLAM II Quick Reference Manual," Pritsker & Associates, Inc., West Lafayette, Indiana.

O'Reilly J.J., Nordlund C. (1989), "Introduction to SLAM II and SLAM-SYSTEM," In: Proceedings of the 1989 Winter Simulation Conference, (Washington, DC, Dec. 4–6, 1989), E.A. MacNair, K.J. Musselman, and P. Heidelberger, (eds.), IEEE, Piscataway, NJ, pp. 178–183.

Ören T.I. (1992), Towards Program Understanding Systems," (Invited Plenary Paper). In: Proceedings of the First Turkish Symposium on Artificial Intelligence and Neural Networks, K. Oflazer, V. Akman, H.A. Gü, U. Halici, (eds.), 25–26 June, 1992, Bilkent University, Ankara, Turkey, pp. 3–12.

Ören T.I., Abou-Rabia O., Luyun L., Birta L., King D. (1989), "MIA: Module Interface Analyzer for Fortran Programs," Technical Report TR-89–55, Computer Science Department, University of Ottawa, Ottawa, Ontario.

Ören T.I., Hamilton I.J., Ören Y.C. (1991), "ORC++: A Graphic Scheme and a Redocumentation Tool for C++," Technical Report TR-91–35, Computer Science Department, University of Ottawa, Ottawa, Ontario.

Ören T.I., Wendt R.N., King D.G., Abou-Rabia O., Birta L.G. (1992). "A Template-Oriented Approach for Simulation Program Understanding," Mathematics and Computers in Simulation, 34: pp. 219–229.

Ören T.I., Zeigler B.P. (1979), "Concepts for advanced simulation methodologies," Simulation 32 (3) pp. 69–82.

Ören T.I., King D. (1989), "OrFor: Organized Representation of Fortran Programs on a Sun Workstation," Technical Report TR-89–16, Computer Science Department, University of Ottawa, Ottawa, Ontario.

Pritsker A.B. (1986), "Introduction to Simulation and SLAM II," 3rd edition, John Wiley and Sons, West Lafayette, Indiana.

Standridge C.R., Pritsker A., Alan B. (1987), "TESS: The Extended Simulation Support System," Halsted Press (John Wiley), New York

Tanimoto S.L. (1987), "The Elements of Artificial Intelligence," Computer Science Press, Rockville, MD.

Wendt R., Ören T.I., Birta L. (1989), "Knowledge-base for SLAM II Understanding System, Volume 1 – Templates," Technical Report TR-89–54 (Prepared for Bell Canada). Simulation Research Group, Computer Science Department, University of Ottawa, Ottawa, Ontario.

Wendt R., Ören T.I., Birta L. (1990), "Knowledge-base for SLAM II Understanding System, Volume 2 – Rule-Base," (Prepared for Bell Canada). Simulation Research Group, Computer Science Department, University of Ottawa, Ottawa, Ontario.

Whitner R.B., Balci O. (1989), "Guidelines for Selecting and Using Simulation Model Verification Techniques," In: Proceedings of the 1989 Winter Simulation Conference, E.A. MacNair, K.J. Musselman, and P. Heidelberger (eds), IEEE, Piscataway, NJ, pp. 559–568.